UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAîTRISE EN GÉNIE ÉLECTRIQUE

PAR
MOURAD ZAKHAMA

IMPLANTATION EN TECHNOLOGIE ITGE (VLSI) D'UN FILTRE ADAPTATIF BASÉ SUR LA LOGIQUE FLOUE POUR L'ÉGALISATION DE CANAUX NON LINÉAIRES

AVRIL 2000

Résumé

Ce travail vise principalement à apporter une contribution pour l'amélioration des systèmes de télécommunication, et plus particulièrement la résolution du problème d'égalisation de canaux. Lors d'une communication numérique, le signal transmis à travers le canal subit une suite de déformation de caractère linéaire et non-linéaire dépendant des caractéristiques du canal. Généralement ces dernières sont inconnues et variables dans le temps. Les déformations introduites sont corrigées au niveau du récepteur par une technique appelée égalisation de canaux. L'égaliseur estime à priori les caractéristiques du canal par la réception d'un paquet de données connues transmises par l'émetteur.

Le but de ce mémoire est l'étude d'une nouvelle méthode d'égalisation de canaux; utilisant la technique de la logique floue, ainsi que de présenter une architecture hautement parallèle à haut débit pour les canaux linéaires ou non-linéaires, variables et invariables dans le temps.

Se référant aux travaux de Wang & Mendel'93 on a pu faire les simulations sur Matlab de l'égaliseur. Notre tâche était d'adapter l'algorithme afin de le préparer pour une implantation en technologie d'intégration à très grande échelle (ITGE – VLSI). La complexité de calcul importante de cet égaliseur rendait difficile son intégration sur

silicium. Ainsi, une étude approfondie de la structure de l'algorithme à intégrer nous a permis d'introduire des simplifications au niveau du filtre et de son adaptation. De plus, grâce à la forme récurrente des équations utilisées, cela nous a permis d'arriver à une architecture systolique pour un égaliseur non-linéaire. D'une première proposition d'architecture, nous avons appliqué des simplifications améliorant de façon significative la surface d'intégration et la vitesse de calcul. L'architecture ainsi dérivée se compose seulement de quatre (4) processeurs élémentaires (PE) et ce peu importe le nombre de fonctions d'appartenance pourvu qu'il soit possible d'utiliser un seul retard sur le signal reçu pour égaliser.

Les résultats des performances de l'architecture systolique proposée ont été évalués d'une part pour une technologie CMOS de 0.5 µm de HP obtenue de la société canadienne de micro-électronique (SCM - CMC) et d'autre part sur une structure logique reconfigurable (FPGA-Field Programmable Gates Array) XC4036EX de la compagnie Xilinx. L'évaluation des performances obtenues après synthèse à l'aide des outils de Synopsys, nous a permis d'atteindre une fréquence d'horloge de 40 MHz et 15 MHz pour les technologies CMOS 0,5 µm et FPGA respectivement.

Enfin, nous avons aussi proposé comment ajouter l'algorithme d'adaptation de type LMS à l'architecture afin d'adapter les coefficients du filtre à logique floue.

Remerciements

Je tiens à exprimer ma profonde gratitude et mes sincères remerciements à mon directeur de recherche, Daniel MASSICOTTE, pour m'avoir fait l'honneur de diriger mon travail de recherche. Je tiens aussi à le remercier pour la confiance et le soutien moral qu'il m'a accordé tout au long de mes études. Son jugement et ses critiques m'ont permis d'acquérir des qualités inestimables en recherche.

La seconde mention ira à mes parents qui m'ont permis de réaliser un rêve, celui de continuer mes études au Canada, et atteindre mes objectifs de carrière. Spécialement mon père, Mansour, qui m'a soutenu moralement et financièrement tout au long de mes études. Je remercie aussi ma mère, Saïda qui n'a pas cessé de me prodiguer ces conseils et son soutien. Ma sincère reconnaissance va aussi à mes frères : Mehdi et Sami ainsi qu'à mes sœurs : Mouna et Marwa. Je témoigne aussi de la gratitude à tous mes oncles, tantes, cousins, et amis pour leur amour, support et compréhension.

Finalement, je tiens à remercier mes coéquipiers du Laboratoire de signaux et systèmes intégrés (LSSI) de l'Université du Québec à Trois-Rivières, ainsi que toute personne qui m'a aidé à réaliser ce mémoire.

Liste des abréviations et symboles

Symboles

e	*Erreur de correction*
s	*Symbole transmis*
W	*Coefficients à adapter*
w	*Poids de la couche cachée*
$\mu_{F_j^y}$	*Degré d'appartenance à une fonction d'appartenance*
p	*Vecteur de sortie du bloque de fuzzification*
θ	*Vecteur déterminant les règles d'inférence*
y	*Sortie du canal*
μ	*Pas d'apprentissage*
λ	*Facteur d'oubli*
η	*Bruit additif du canal*
$\tilde{\bullet}$	*Signal bruité*
$\bar{\bullet}$	*Signal déformé linéairement*
$\hat{\bullet}$	*Estimé de*

Abbreviations

ASIC	***A**pplication **S**pecific **I**ntegrated **C**ircuit*
BER	***B**it **E**rror **R**ate*
CLB	***C**ontrol **L**ogic **B**locs*
CSA	***C**arry **S**ave **A**dder*
DFE	***D**ecision **F**eedback **E**qualizer*
DSP	***D**igital **S**ignal **P**rocessor*
FA	***F**ull **A**dder*
FPGA	***F**ield **P**rogrammable **G**ate **A**rray*
IOB	***I**nput **O**utput **B**locs*
ITGE	***I**ntégration à trés **G**rande **É**chelle*
LAN	***L**ocal **A**rea **N**etwork*
LMS	***L**east **M**ean **S**quares*
PE	***P**rocesseur **É**lémentaire*
RLS	***R**ecurcive **M**ean **S**quares*
VHDL	*Very Large Scale Integration Circuit **H**ardware **D**escription **L**anguage*

Table des matières

Table des figures

Liste des Tableaux

Chapitre 1

Introduction

On peut estimer que l'histoire des télécommunications a commencé en 1832 avec la découverte par le physicien américain Morse du mode de transmission codé. Parallèlement la téléphonie s'est développée. Marconi réalisa en 1899 la première liaison télégraphique par ondes hertzienne entre la France et l'Angleterre. En 1962 le satellite Telstar a permis la première liaison de télévision transocéanique. Ceci n'était qu'un point de départ pour le développement des moyens de communications. Le besoin de plus en plus croissant de la transmission temps réel et l'augmentation du volume des données transmises, va nous amener à améliorer la qualité et la quantité des données qui voyagent à travers les supports physiques. De nos jours on a tendance à faire le traitement numérique de l'information.

La théorie de la transmission de l'information a connu un formidable essor depuis que les travaux de Shannon (1948) ont démontré la possibilité théorique d'une transmission fiable.

Elle fait appel à des nombreuses disciplines : codage source, codage canal, égalisation et, plus généralement, à ce qui est convenu d'appeler la théorie de l'information. Deux grandes classes de techniques permettent d'assurer la fiabilité d'une transmission : le codage canal qui vise à coder le message émis de telle sorte que le récepteur soit à même de corriger la plupart des erreurs de transmission, et l'égalisation, dont l'objectif est d'exploiter au mieux la bande passante du média (canal de transmission).

Pour l'égalisation de canaux, théoriquement, l'algorithme de Viterbi [PRO95] permet d'avoir la transmission la plus fiable possible. Cependant, deux points limitent son usage : d'une part il présente une forte complexité calculatoire et d'autre part, il suppose que le récepteur connaît les distorsions dues à la transmission alors que les canaux de transmission réels sont inconnus et variables au cours du temps.

La méthode qu'on va développer dans ce mémoire est celle d'un égaliseur de canaux adaptatifs pour les canaux linéaires et non-linéaires basés sur la logique floue.

1.1 Problématique de recherche

La plupart des canaux de communication, entre autres les canaux téléphoniques, modems et certains canaux de radio, sont caractérisés par le passage d'un signal dans un canal de communication à bande passante limitée. De ce fait, si la vitesse de transmission du message est supérieure à la fréquence de coupure du canal, le signal transmis à travers ce canal va subir une distorsion qui dépend des caractéristiques de ce dernier. Pour augmenter le débit de transmission en télécommunication plusieurs méthodes ont été utilisé, telle que la compression de données [MAR95]. Ces dernières n'ont pas satisfait les exigences de plus en plus croissantes en terme de débit. Deux solutions sont possibles : soit changer le canal

de transmission pour augmenter sa bande passante ou bien prévoir un système de codage à l'émission et de correction à la réception. La première solution est assez coûteuse car les supports de transmission sont déjà existants et leur remplacement est difficile en plus il y a des supports de transmission qu'on ne peut pas changer entre autre l'air pour la transmission hertzienne. La seconde solution est celle qui permet d'utiliser les mêmes installations pour mieux servir l'utilisateur. Cette dernière consiste à implanter au niveau du récepteur un système qui permet de récupérer l'information ou signal perturbé ou modifié, c'est ce qu'on appelle système d'égalisation de canaux. L'égaliseur a pour rôle d'estimer le signal transmis à partir du signal au récepteur. En absence de bruit l'égaliseur est simplement un filtre inverse dont le rôle est de rendre plate la réponse impulsionnelle de l'ensemble canal & égaliseur. Par contre lorsque le canal est bruité un simple filtre inverse amplifie considérablement le bruit surtout dans les régions où le signal utile est faible. Ce qui donne un signal totalement différent de celui qui était à l'entrée du canal. Pour remédier à ce problème on a souvent utilisé des méthodes d'égalisation des canaux basées sur le filtre linéaire transverse [HAY96]. La procédure d'étalonnage où la présence d'un canal non invariant et non stationnaire oblige une adaptation du modèle. Plusieurs algorithmes d'adaptations sont possible mais seulement quelques-uns sont envisageables pour une intégration en technologie d'intégration à trés grande échelle (ITGE - en anglais VLSI - Very Large Scale Integration). Les méthodes les plus fréquemment utilisées sont l'algorithme du gradient stochastique (LMS - Least Mean Square) et l'algorithme des moindres carrés récursifs (RLS - Récursive Least Squares).

Le schéma de principe de la chaîne de transmission numérique est représenté sur la figure 1.1 [GLA96]. On peut distinguer : la source de message, le milieu de transmission

(ou canal de transmission), et le destinataire qui sont les données du problème. Le codage et le décodage de source, le codage et le décodage de canal, l'émetteur et le récepteur sont les degrés de liberté du constructeur pour réaliser le système de transmission.

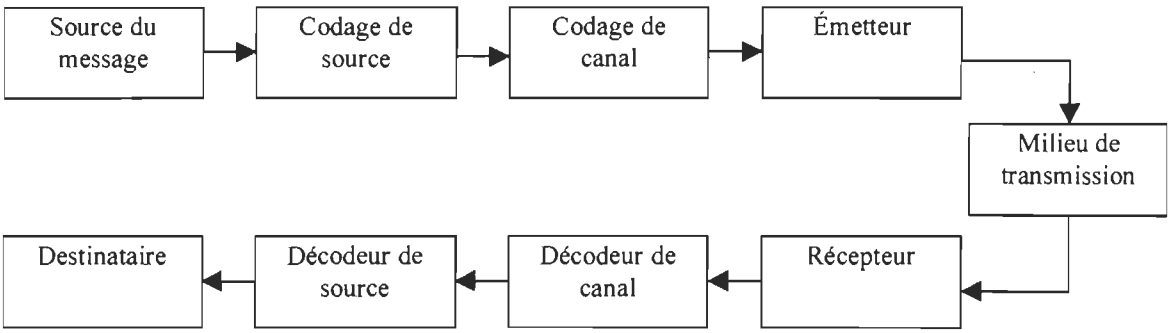


Figure 1.1 : Principe d'une chaîne de transmission numérique

Le problème à résoudre dans ce projet est de proposer une méthode d'égalisation convenant aux canaux linéaires et non linéaires et pouvant satisfaire les contraintes et critères d'implantation en technologie ITGE. Pour cela, on va étudier la possibilité d'implanter un égalisateur adaptatif en se basant sur la technique de la logique floue. L'égaliseur à base de la logique floue, se caractérise par le fait qu'il présente des équations récurrentes menants à la possibilité d'une architecture ITGE hautement parallèle. En outre la logique floue présente un processus non linéaire qui se prête bien à la correction de canal de communication non linéaire. De plus, l'étude de la structure de l'algorithme par logique floue révèle la présence de plusieurs termes nuls dans l'étape de fuzzification, d'où la possibilité d'avoir une simplification significative de la complexité de calcul et de la surface d'intégration de l'architecture associée.

1.2 Objectifs

Le but de ce travail est de proposer un algorithme adaptatif performant pour les canaux linéaires et non linéaires. Il s'agit ensuite de simplifier et optimiser les différentes parties de l'algorithme pour pouvoir l'implanter en technologie VLSI (FPGA, ASIC). La recherche du compromis débit de calcul et surface d'intégration va nous permettre d'arriver avec une architecture numérique hautement parallèle. L'intégration en technologie CMOS ou sur FPGA nous permettra d'évaluer les performances de l'architecture en terme de latence, débit et surface d'intégration.

1.3 Méthodologie de recherche

A la suite d'une recherche bibliographique sur les algorithmes et architectures ITGE pour l'égalisation de canaux non linéaires, nous allons pouvoir justifier le choix de notre modèle basé sur la logique floue. En se basant sur les travaux de Wang & Mendel[1] [WAN93], qui ont proposé un égalisateur adaptatif à base de la logique floue avec adaptation des paramètres par un algorithme LMS ou RLS, nous allons étudier la possibilité d'implanter ce filtre en technologie ITGE.

Nous aurons à valider les performances de ce filtre par des résultats de simulation dans l'environnement Matlab® et par la suite il faudra étudier les caractéristiques du filtre mais sans adaptation des paramètres. Ceci va nous permettre d'évaluer la robustesse de ce type d'égaliseur vis-à-vis de la variation du niveau de bruit. Lors de l'étude de l'implantation de cet égaliseur, notre tâche sera de modifier l'algorithme afin de faciliter son implantation.

1 Article publié dans le journal IEEE : « fuzzy adaptative filter with application to non-linear channel equalization » c'est l'article sur lequel se base notre étude.

On sait déjà que l'algorithme RLS présente une trés grande complexité de calcul. Par conséquent on va étudier l'utilisation de l'algorithme LMS pour une implantation ITGE.

Nous proposerons une architecture systolique en considérant le canal comme étant invariant et par la suite nous proposerons une architecture pour le modèle adaptatif. La proposition d'une telle architecture exigera la recherche du compromis débit de calcul et surface d'intégration. La validation de l'architecture proposée se fera par la modélisation, la simulation et la synthèse du code VHDL avec les outils de Mentor Graphics® et Synopsis®. La synthèse des résultats fera état des performances de l'algorithme et de l'architecture systolique proposée. Un exemple de mise en œuvre de l'égaliseur proposé sera réalisé dans un circuit FPGA-DSP.

1.4 Structure du rapport

Ce mémoire se divise essentiellement en quatre chapitres (voir figure 1.2). Le premier chapitre sera consacré à expliciter la problématique de recherche, les objectifs ainsi que la méthodologie suivie tout au long de la réalisation de ce travail.

Dans le second chapitre, on commencera par définir le canal de transmission dans la section 2.1. par la suite on définira la problématique d'égalisation de canaux à la section 2.2. Au niveau de la section 2.3 on présentera les algorithmes d'égalisation de canaux que ce soit linéaire ou non-linéaire. Ensuite on présentera au niveau de la section 2.4 les différentes architectures retrouvées dans la littérature. La section 2.5 portera sur la justification du choix d'une technique pour l'égalisation de canaux, celle utilisant la logique floue. Cette technique sera détaillée dans le chapitre 3. À la section 3.2 on expliquera comment

appliquer la logique floue pour résoudre le problème d'égalisation de canaux. Enfin, au niveau de la section 3.3, on essayera d'apporter les simplifications nécessaires à l'algorithme proposé afin de le préparer à une implantation en technologie ITGE.

Le chapitre 4 portera sur les architectures proposées et leurs implantations. On commencera par une étude de quantification au niveau de la section 4.2, laquelle nous permettra de fixer le nombre de bits à utiliser dans l'architecture. Ensuite on proposera une architecture systolique dans la section 4.3. enfin on étudiera, dans la section 4.4, deux types d'implantation : celle pour un ASIC en technologie CMOS 0.5 µm dans la section 4.4.4 et celle d'une implantation en technologie FPGA de la compagnie Xilinx dans la section 4.4.5. et on terminera avec une conclusion générale.

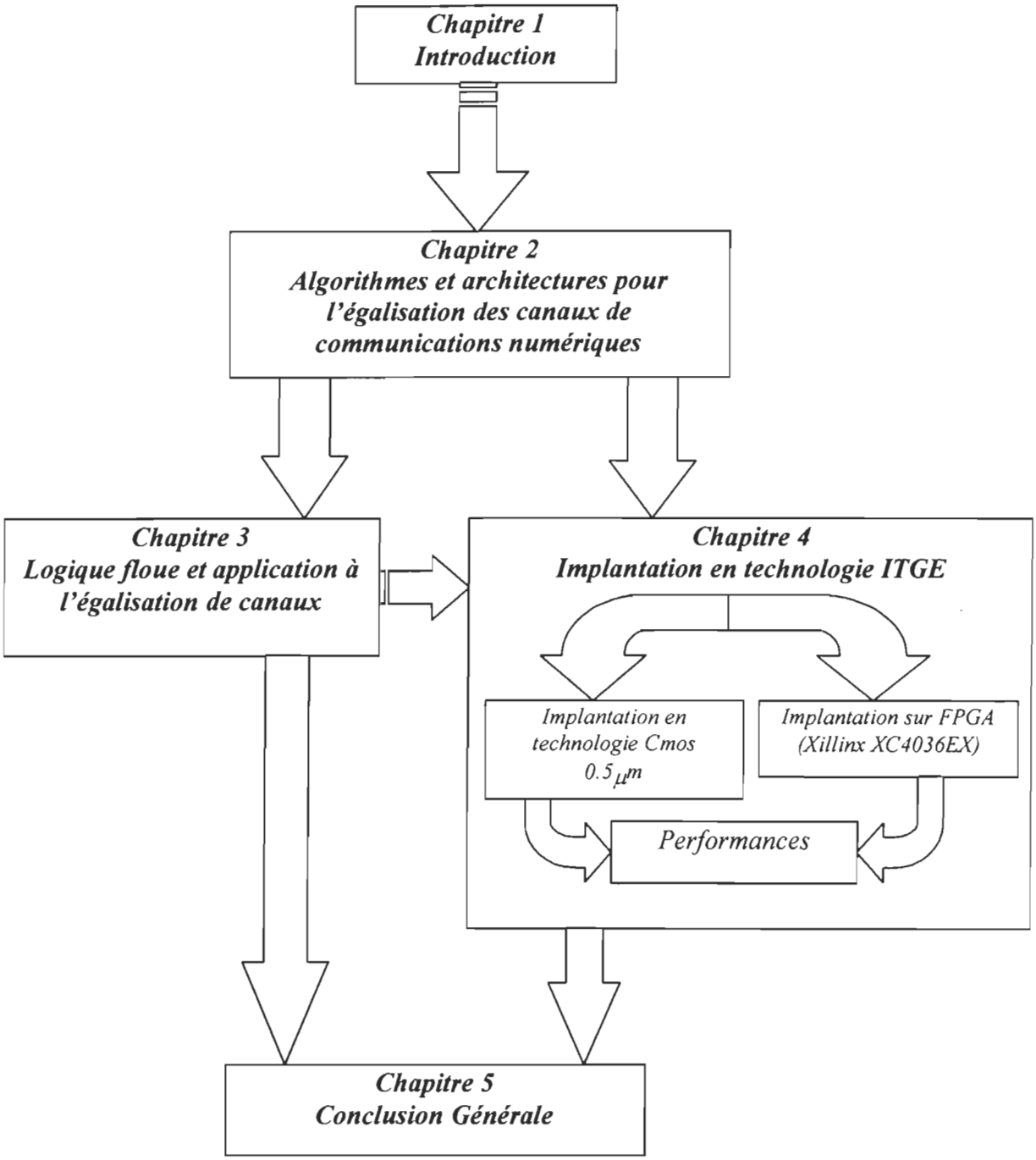


Figure 1.2 : Structure du rapport

Chapitre 2

Algorithmes et architectures pour l'égalisation de canaux

Il existe plusieurs formes sous lesquelles une information peut être présentée. On peut faire la transmission de la voix d'un signal vidéo, par conséquent l'information à transmettre peut être de type numérique ou analogique. On a toujours tendance à ramener à une représentation numérique par un convertisseur analogique à numérique (A/N). D'une manière générale et comme c'est présenté sur la figure 2.1, les systèmes de communication numérique commencent par un convertisseur A/N qui permet de représenter l'information sous forme binaire, ensuite on traite le signal numérisé pour lui appliquer une méthode de codage ou de compression. Ceci se passe dans le bloc encodeur. Le dernier élément qui permettra d'avoir le module d'émission est le modulateur. Il a pour rôle de convertir le signal numérique en un signal analogique pouvant être transmis dans un canal qui constitue

le médium de communication. Il peut s'agir d'un câble coaxial, bifilaire, une fibre optique ou encore l'air. Une fois le signal arrivé au récepteur, on commence par démoduler le signal afin d'obtenir l'information transmise. Au niveau du démodulateur il y aura un système qui permettra d'annuler ou de corriger les effets de distorsion apportés par le canal. C'est ce qu'on appelle l'égalisation de canaux.

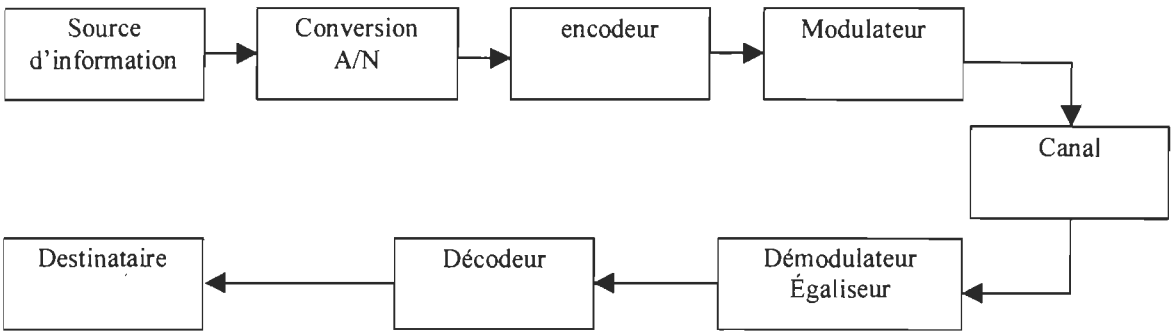


Figure 2.1 : schéma bloque d'un système de communication numérique

On trouve dans la littérature plusieurs algorithmes pour l'égalisation de canaux linéaires et non linéaires. De plus, l'intégration en technologie ITGE est trés rare. Ce chapitre couvrira le domaine de recherche scientifique au niveau algorithmique et architectural d'égalisation de canaux linéaires et non linéaires. Par conséquent on va exposer certaines méthodes trouvées dans la littérature permettants de résoudre le problème d'égalisation de canaux linéaires et non linéaires. On va commencer par définir le canal de transmission dans le paragraphe 2.1, ensuite on expliquera la problématique d'égalisation de canaux à la section 2.2. La section 2.3 sera consacrée aux algorithmes d'égalisation de canaux. Ensuite on survolera le domaine de l'intégration ITGE de ce type d'algorithmes à la section 2.4. enfin, on aura à justifier l'utilisation de la logique floue pour la suite du travail.

2.1 Canal de transmission

2.1.1 Définition du canal

La plupart des canaux de communication entre autres les canaux téléphoniques, certains canaux de radio, sont caractérisés par une bande passante présentant des évanouissements d'amplitude et des non-linéarités de phase. Par conséquent, pour l'étude de la distorsion apportée par le canal, on peut caractériser chaque canal par sa réponse en fréquence C(f) comme suit :

$$C(f) = A(f).e^{j.\theta(f)} \tag{2.1}$$

où A(f) est la réponse en amplitude du canal, θ(f) est la réponse en phase du canal.

Une autre caractéristique souvent utilisée pour caractériser un canal de transmission est l'enveloppe de délai définie comme suit:

$$\tau(f) = -\frac{1}{2\pi}\frac{d\theta(f)}{df} \tag{2.2}$$

Un canal est dit idéal si dans la bande de fréquence du signal transmis la réponse en amplitude du canal A(f) est constante et la réponse, θ(f), est linéaire (ou τ(f)=constante) pour toutes les fréquences. Si A(f) n'est pas constante la distorsion subit par le signal est dite d'amplitude alors que si τ(f) n'est pas constante la distorsion est dite de délai ou de phase. En résumé les termes A(f) et θ(f) ou τ(f), définissent les caractéristiques du médium de transmission qui influent sur le signal qui traverse le canal en le distordant, ce qui donne un BER≠ 0 sur le signal de sortie du canal. En plus de cette déformation apportée au signal

s'ajoute le bruit qui sera mesuré comme étant le rapport entre le signal transmis et le bruit ce qui définit le SNR.

2.1.2 Modélisation du canal

D'après la théorie du codage, un canal de transmission inclut toutes les fonctions situées entre la sortie du codeur du canal et l'entrée du décodeur [GLA96], voir figure 2.2.

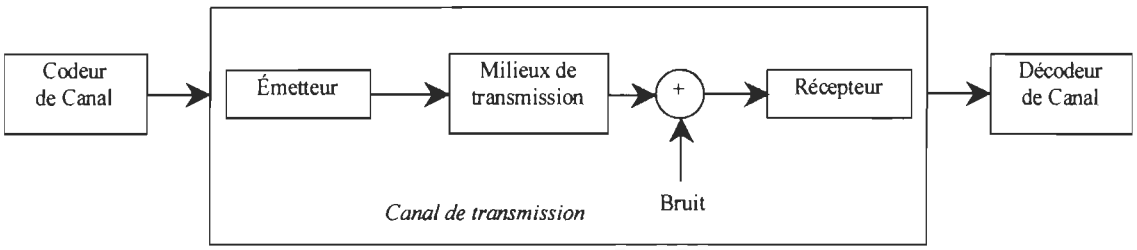


Figure 2.2 : Le canal de transmission[GLA96]

Les distorsions ou les déformations apportées au signal original peuvent parvenir de l'émetteur, du milieu de transmission, du bruit additif et du récepteur. L'émetteur est le bloque qui permet de transformer le signal numérique en un signal analogique, par l'ajout de la porteuse, ce qui le rend transportable par un canal. Le milieu de transmission peut être de l'air, un câble coaxial ou bifilaire, une fibre optique..., les caractéristiques du milieu introduisent des déformations sur le signal. Au niveau du récepteur on se trouve avec un signal analogique déformé dont la démodulation va encore introduire des distorsions. Toutes ces déformations subites par le signal au cours de son trajet depuis le codeur jusqu'au décodeur sont apportées par le canal de transmission. Ces distorsions peuvent être de caractère linéaire ou non-linéaire, stationnaires ou variables dans le temps. Dans le langage mathématique on peut remplacer la figure 2.2 par la figure 2.3. Le canal est

essentiellement composé de la partie linéaire qui reflète la réponse impulsionnelle du canal, généralement modélisée par un filtre passe bas, et une partie non-linéaire due aux autres composantes du canal de communication. En plus on aura un bruit additionnel inévitable η s'additionnant au signal y_k pour donner le signal $\tilde{y}_k$ qui constitue la sortie du canal de transmission. Ce signal est l'image du signal s_k après avoir traversé le canal de transmission.

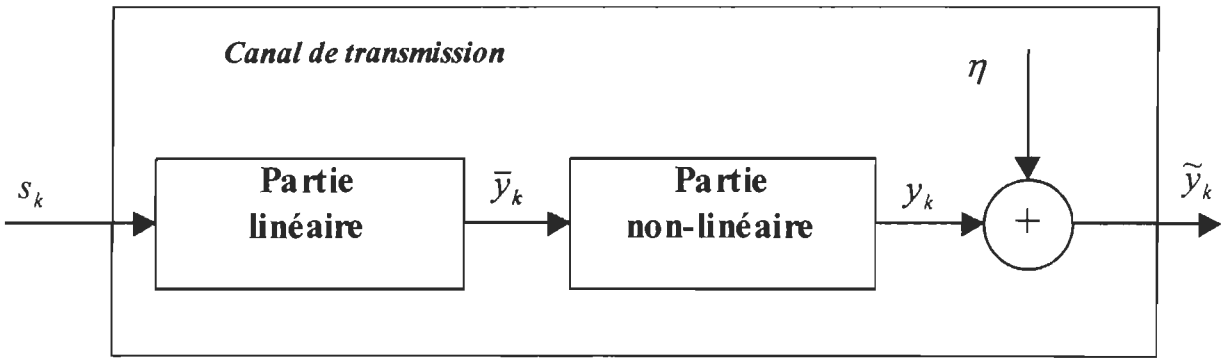


Figure 2.3 : Modélisation d'un canal de communication

2.2 Égalisation de canaux

2.2.1 Principe

L'origine du terme égalisation se comprend aisément dans le domaine des fréquences. Dans un égaliseur graphique de chaîne HI-FI [BRO97], des curseurs permettent d'ajuster des filtres passe bande autour de certaines fréquences. En réglant correctement les différents curseurs, l'utilisateur modifie à sa guise le spectre du signal issue de la source. Avant de parvenir à l'oreille de l'auditeur, le son est déformé par la réponse de la pièce d'écoute. La mise en cascade des déformations spectrales dues à la pièce et celles dues à l'égaliseur

graphique correctement réglé, permet d'obtenir une réponse globale plate sur la bande audio. La réponse de la chaîne liant la source à l'auditeur a été égalisé.

Dans le domaine des télécommunications, les données émises traversent un canal de transmission et les différents dispositifs électroniques associés à l'émetteur et au récepteur. Le rôle de l'égaliseur est alors de réduire au mieux les distorsions apportées par ces éléments. Pour compenser ces distorsions on utilise un filtre linéaire avec des paramètres ajustables suivant les caractéristiques du canal. Ces filtres ajustables sont appelés des égaliseurs ou encore égaliseurs de canaux. Pour les canaux avec une réponse en fréquence inconnue mais invariante dans le temps, on mesure les caractéristiques du canal et on ajuste en fonction les paramètres de l'égaliseur. Ces paramètres restent fixes durant la transmission des données. Ce type d'égaliseur est appelé égaliseur pré-établit (preset equalizer).

Lorsque le canal de transmission n'est plus invariant dans le temps, les techniques adaptatives de traitement permettent de concevoir des algorithmes estimants en permanence les paramètres pour l'égaliseur en minimisant un certain critère. On parle alors d'égalisation adaptative. Les algorithmes d'adaptation utilisés sont dits à décision rétroactive et les plus fréquemment utilisés sont les algorithmes LMS et RLS.

2.2.2 Égalisation adaptative

La différence avec les égaliseurs pré-établit est que les paramètres de l'égaliseur adaptatif ne sont pas fixes, mais ils seront adaptés par l'envoi d'un paquet de données connues $s(k)$. La figure 2.4 présente un égaliseur adaptatif, $s(k)$ est la séquence de données provenant de l'émetteur traversant un canal qui y introduit des déformations dues à ces caractéristiques

linéaires ou non-linéaires. La sortie du canal est le signal $r(k)$ corrompu par un bruit additif $\eta(k)$ nous donne le signal $\tilde{r}(k)$. A partir de l'erreur de reconstitution $e(k-d)$ les paramètres du filtre sont optimisés par un algorithme de minimisation d'erreur. L'adaptation des paramètres prendra un certain nombre de cycles afin de converger vers un minimum d'erreur de correction. Une fois que l'égaliseur est adapté au canal les paramètres de l'égaliseur sont conservés constants pour toute la durée de la communication, c'est le cas de la communication par modem. Par contre, si le canal varie l'adaptation doit assurer la poursuite des variations du canal par l'ajustement des paramètres de l'égaliseur, ce qui en général ne cause pas de problème pour de faibles variations. Dans le cas d'une variation brusque ou rapide du canal (milieu hostile), le temps d'adaptation devient critique et dépendra de l'algorithme utilisé et de sa mise en œuvre. Le signal de sortie de l'égaliseur $\hat{x}(k-d)$ passe ensuite par une fonction de décision pour donner le signal $\hat{s}(k-d)$.

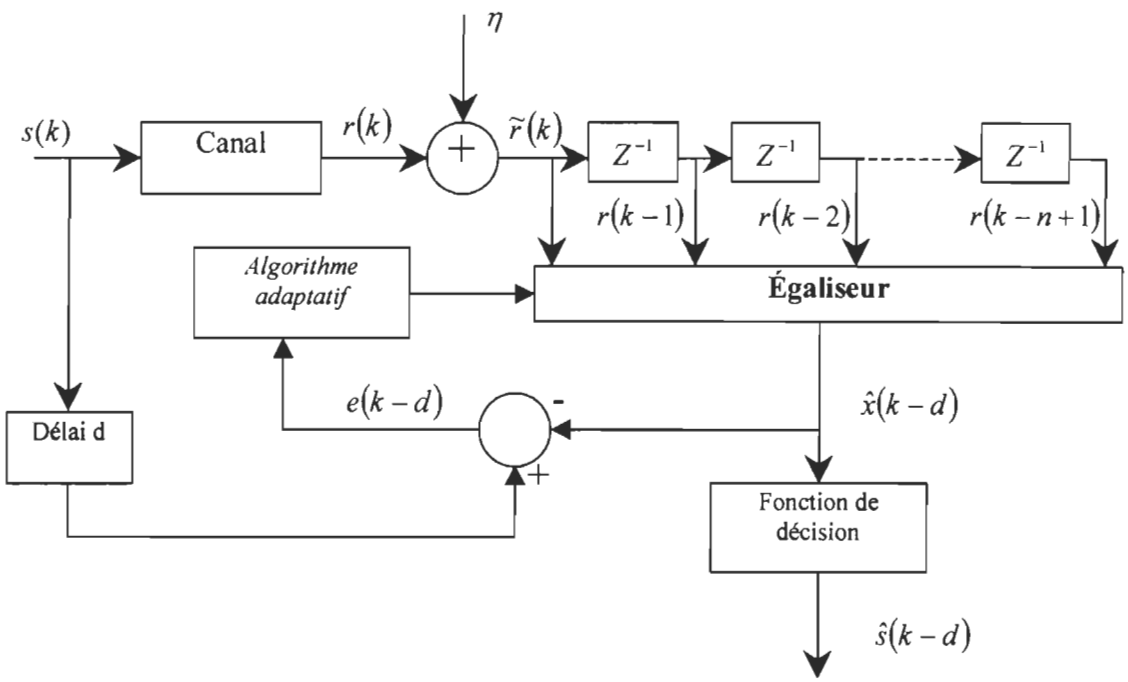


Figure 2.4 : Système d'égalisation de canaux adaptatif.

2.2.3 Égalisation autodidacte

Il n'est pas toujours possible de disposer d'une séquence d'apprentissage de la part de l'émetteur, De ce fait un autre type d'égalisation est utilisé, c'est ce qu'on appelle auto égalisation ou égalisation aveugle en anglais c'est "blind equalization". Le but de l'égalisation aveugle est de pouvoir reconstituer les données émises $x(k)$ à partir de la seule observation de la sortie du canal $u(k)$ (voir figure 2.5). On trouve principalement trois types d'algorithmes [PRO95] : L'égalisation aveugle basée sur le maximum de similitude, et les statistiques sur le signal d'ordre deux ou plus.

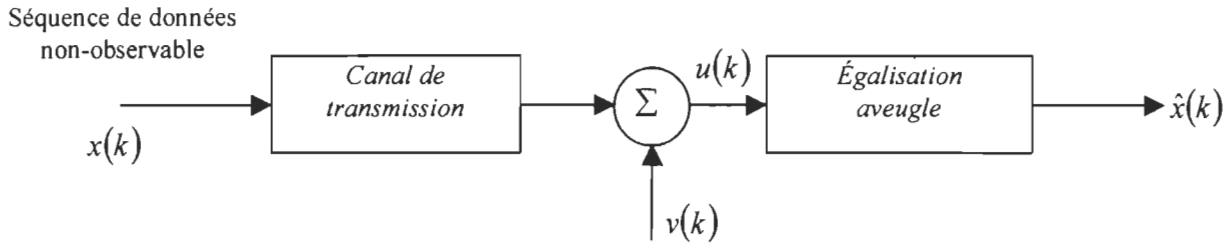


Figure 2.5 : Schéma bloc d'un égaliseur aveugle[HAY96]

2.3 Algorithmes pour l'égalisation de canaux

2.3.1 Algorithmes linéaires pour l'égalisation de canaux

L'algorithme le plus couramment utilisé est le filtre linéaire transverse souvent appelé filtre à réponse impulsionnelle finie (RIF - FIR - Finit Impulse Response) présenté sur la figure 2.6. Il est défini par la relation de convolution linéaire suivante :

$$y(k) = \sum_{n=0}^{M-1} w(n)u(k-n) \tag{2.3}$$

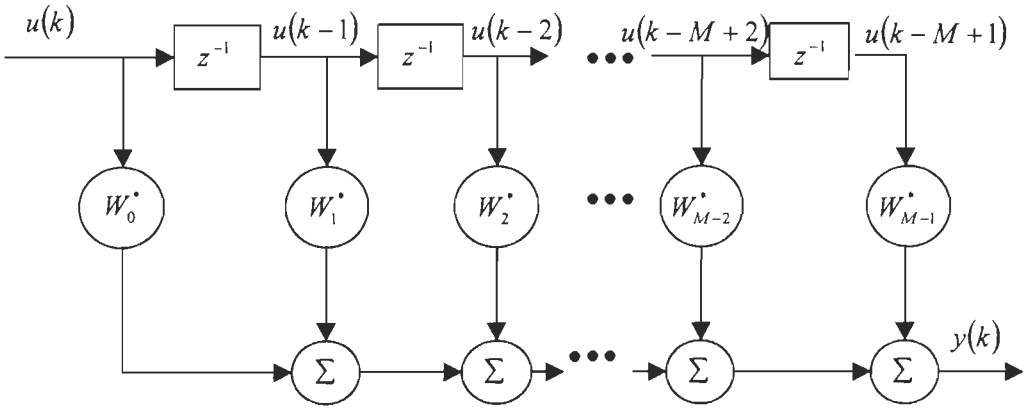


Figure 2.6 : Filtre transverse [HAY96]

Les algorithmes d'adaptation des coefficients W les plus souvent utilisés sont LMS et RLS [HAY96]. Malheureusement, l'équation (2.3) ne s'applique que pour des systèmes de communication ayant un canal linéaire. L'égalisation de canaux linéaires se fait bien avec ces derniers, sauf que dans la pratique il est trés rare de traiter des canaux linéaires. Le cas le plus général est celui des systèmes non linéaires. La logique floue [WAN93], [SAR95], [LEE94] et les réseaux de neurones [VID99], [KEC94] comme le montre plusieurs exemples dans la littérature, sont de plus en plus utilisés pour la modélisation de systèmes complexes. Ils s'apprêtent aussi bien pour les systèmes linéaires que non-linéaires. Le tableau 2.1 présente les principales équations permettants de modéliser le filtre LMS et RLS.

Tableau 2.1 : Sommaire des méthodes classiques [HAY93]

LMS	
$\hat{x}(k) = \mathrm{w}^T \tilde{\mathrm{y}}(k)$	(2.4)
$e(k) = s(k-d) - \hat{x}(k)$	(2.5)
$\mathrm{w}(k+1) = \mathrm{w}(k) + \mu\, \tilde{\mathrm{y}}(k) e(k)$	(2.6)

RLS	
$\hat{x}(k) = \mathrm{w}^T \tilde{\mathrm{y}}(k)$	(2.7)
$P(k) = \frac{1}{\lambda} P(k-1) - \frac{1}{\lambda} K(k) \tilde{\mathrm{y}}^T(k) P(k-1)$	(2.8)
$K(k) = \frac{1}{\lambda} * \frac{P(k-1)\tilde{\mathrm{y}}(k)}{1 + \frac{1}{\lambda} \tilde{\mathrm{y}}^T(k) P(k-1) \tilde{\mathrm{y}}(k)}$	(2.9)
$\mathrm{w}(k) = \mathrm{w}(k-1) + K(k) e(k)$	(2.10)
$e(k+1) = s(k-d) - \hat{x}(k)$	(2.11)

2.3.2 Algorithmes non-linéaires pour l'égalisation de canaux

Les principaux égaliseurs non linéaires sont basés sur la structure DFE adaptative (DFE:"Decision Feedback Equalizer"). La structure dotée d'une partie transversale et d'une partie récursive avec décision dans la boucle de retour est appelée structure à décision dans la boucle ou à retour de décision (DFE). Cette structure est tout particulièrement adaptée aux systèmes numériques de communication. Une structure DFE permet d'égaliser des canaux beaucoup plus sévères qu'un simple filtre transverse linéaire.

Les coefficients du filtre transverse, de longueur finie, sont périodiquement actualisés de façon à minimiser un certain critère. Cette actualisation des coefficients est généralement réalisée après chaque décision. L'optimisation de ces paramètres se fait avec un algorithme linéaire tel que LMS ou RLS.

Considérons le cas d'un DFE utilisant LMS pour l'adaptation.

$$\hat{x}(k) = \sum_{n=1}^{N} w(n) v(n) \qquad (2.12)$$

$$\hat{s}(k) = D(\hat{x}(k)) \tag{2.13}$$

Où w sont les K coefficients du filtre et v les entrées. La fonction D est une fonction non-linéaire de décision. La mise à jour des coefficients peut se faire à partir de l'équation :

$$W(k+1) = W(k) + \mu v(k) e(k) \tag{2.14}$$

La figure 2.7 présente un exemple d'égaliseur de canaux non linéaires basé sur la logique floue utilisant LMS ou RLS pour l'adaptation des paramètres. Cet égaliseur va être détaillé dans les prochains chapitres. En effet c'est l'algorithme que l'on va implanter en technologie ITGE.

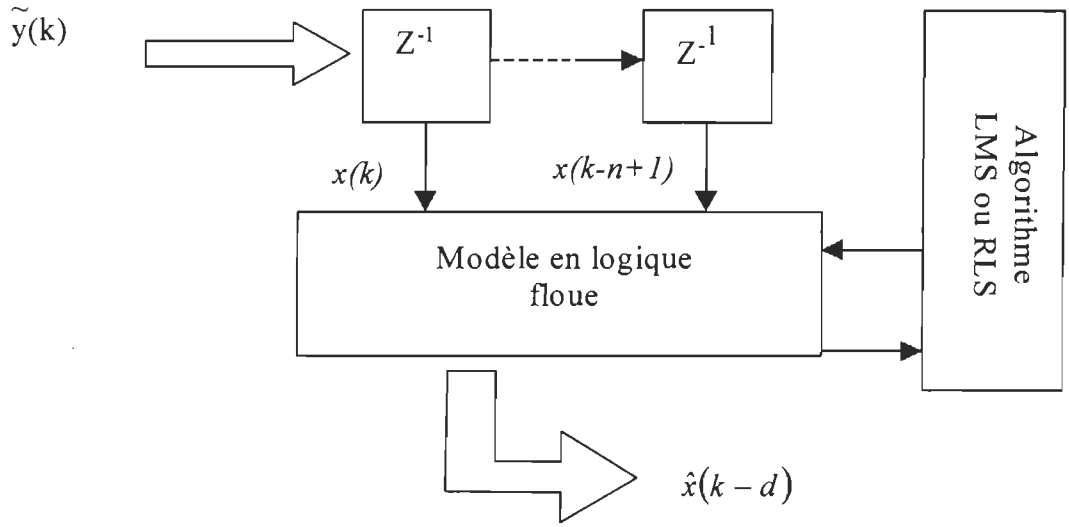


Figure 2.7 : Schéma simplifié de l'égaliseur à base de logique floue

$\tilde{y}(k)$ est l'échantillon d'entrée à l'instant k

$x(k)$ est le signal d'entrée à l'égaliseur à l'instant k

$x(k-n+1)$ est le signal d'entrée à l'égaliseur avec un retard n

$\hat{x}(k-d)$ est la sortie du canal après un délai d

2.4 Architecture pour l'égalisation de canaux

Plusieurs algorithmes ont été développés pour l'égalisation de canaux linéaires et non linéaires, cependant l'implantation en technologie ITGE reste limitée. La plupart des égaliseurs implantés sont analogiques. On peut citer l'exemple de processeur analogique réalisé par [CHO93] pour un système de réception.

Xue et Hu ont présenté un réseau de neurones partiellement connecté pour le système de GSM (Global System for Mobile communication) en Europe [XUE96]. Cette approche convient à une implantation ITGE. Vidal a présenté un réseau de neurones basé sur l'utilisation d'une fonction canonique linéaire par morceaux (CLM - en anglais Piecewise Linear) pour l'égalisation de canaux non linéaires [VID99]. Une autre architecture pour l'égalisation de canaux basée sur la logique floue a été développé par [PAT98].

2.5 Justification d'une intégration en technologie ITGE

Dans le domaine de la communication numérique plusieurs projets ont été fait pour améliorer la qualité et le volume des données transmises. Parmi les éléments constituants le système de télécommunication, c'est l'égalisation de canaux qui permet à l'utilisateur d'aller au-delà de la bande passante du canal. Les implantations pratiques des égaliseurs ont été souvent faites analogiquement. De nos jours nous tendons vers l'utilisation des circuits numériques qui présentent une trés grande fiabilité et robustesse au bruit. L'intégration ITGE d'égaliseurs de canaux numériques est encore au stade de la recherche. C'est un sujet d'actualité et rare sont les articles publiés. L'application de la logique floue à la télécommunication est encore un domaine ouvert et qui mènera à des solutions intéressantes dans les prochaines années. L'intégration d'un égaliseur de canaux à base de

logique floue est donc un sujet original. En plus la logique floue s'apprête bien aux systèmes non linéaires. L'algorithme d'égalisation de canaux basé sur la logique floue proposé par Wang et Mandel [WAN93] (Annexe F) s'applique aussi bien pour les canaux linéaires que non-linéaires et l'étude de sa structure révèle la présence d'équations récurrentes pouvant conduire à une implantation ITGE avec une architecture hautement parallèle. En plus, comme on va le voir dans le chapitre 3, la simulation de l'algorithme à base de logique floue nous a permis d'obtenir des résultats trés satisfaisants. C'est pour ces raisons qu'on a choisi de faire l'étude de cet algorithme pour l'implanter en technologie ITGE.

Chapitre 3

Égaliseur basé sur la Logique floue

Les démarches fondamentales de l'activité industrielle et économique, telle la conception des produits, la gestion des systèmes ou la prise de décision, posent des problèmes de complexité croissante, où, pour certains d'entre eux, une différence majeure tient à ce que les informations fournies ne sont pas précises ou ne peuvent être traitées dans un cadre probabiliste. Face à cette difficulté, les approches numériques, pourtant bien développées (les mathématiques de la décision ou de la théorie de la commande) ou symboliques, (l'intelligence artificielle et les systèmes à base de connaissances), ce sont avérées d'une efficacité limitée. Ce qui a poussé les scientifiques à s'intéresser à la formalisation des connaissances subjectives. Un pas décisif semble avoir été fait en 1965, avec le concept d'ensemble flou, proposé par Lotfi ZADEH [ZAD65], professeur de l'université de Californie à Berkeley.

De manière générale, la résolution d'un problème d'automatisme demande tout d'abord une modélisation mathématique du système à piloter, le plus souvent sous forme d'équations différentielles, permettants de calculer par exemple la commande optimale. Dans la pratique, cependant, il est rare de définir un modèle mathématique exact et simple à exploiter. Ces difficultés ont conduit Zadeh à proposer un moyen de décrire les relations entre les variables d'un système, et à publier un article d'une quinzaine de pages intitulé « fuzzy sets » (ensembles flous) [ZAD65], dix ans après il publia « the concept of linguistic variable and it's application to approximate reasonning » [ZAD75]. Ce qui a amené à introduire des concepts constituants la logique floue. Cette nouvelle technique de traitement des systèmes donne une approche plutôt pragmatique, permettant d'inclure aussi des expériences acquises par des opérateurs. Une série de travaux publiée par la suite ont contribué à l'échafaudage de cette théorie sur laquelle travaillent aujourd'hui plusieurs équipes universitaires et industrielles.

Les premières applications industrielles de la logique floue datent des années 70. En 1975 E.H Madani expérimenta un régulateur floue qu'il perfectionne dans les années suivantes. Cependant le vrai essor applicatif et médiatique n'est apparu que vers la fin des années 80, en grande partie grâce à l'intérêt que lui à porté le Japon. Aujourd'hui, la logique floue est de grande actualité, il s'agit d'une nouvelle méthode de traitement et de prise de décisions. Elle est surtout utilisée dans le domaine du réglage et de la commande de processus industriels.

Récemment on commence à entendre parler de la logique floue dans le domaine de la télécommunication et de l'informatique. Parmi ces applications on peut citer la réalisation d'un processeur RISC (Reduced Instruction Set Computer) pour la logique floue [EDD95],

en outre dans le domaine de la télécommunication Wang & Mandel ont proposé un algorithme pour l'égalisation de canaux linéaires et non linéaires à base de la logique floue [WAN93].

Dans ce chapitre on va commencer par présenter le vocabulaire utilisé, donner les notions de base de la logique floue, à savoir la notion d'ensemble flou, opérateurs logiques, variables linguistiques, et fonctions d'appartenance ensuite on détaillera les différents blocs constituants un système à base de logique floue : la fuzzification, les règles d'inférence et la défuzzification et on terminera cette première section par les caractéristiques et cas d'application de cette logique. La section 3.2 de ce chapitre sera consacrée à l'application de la logique floue à l'égalisation de canaux, entre autre la présentation du système d'égalisation et sa formalisation. Par la suite on présentera les résultats de simulation dans l'environnement Matlab® de l'égaliseur ainsi formulé. Et on terminera à la section 3.3 par une formulation d'un algorithme en vue d'une implantation en technologie ITGE.

3.1 Logique floue

3.1.1 Notions de base

Le raisonnement par logique floue ressemble en quelque sorte au raisonnement humain, la figure 3.1 en donne un exemple : une personne dans une chambre, ressent que la température est un peu élevée. Dépendamment de la température de la chambre la personne va ouvrir la fenêtre à un certain degré.

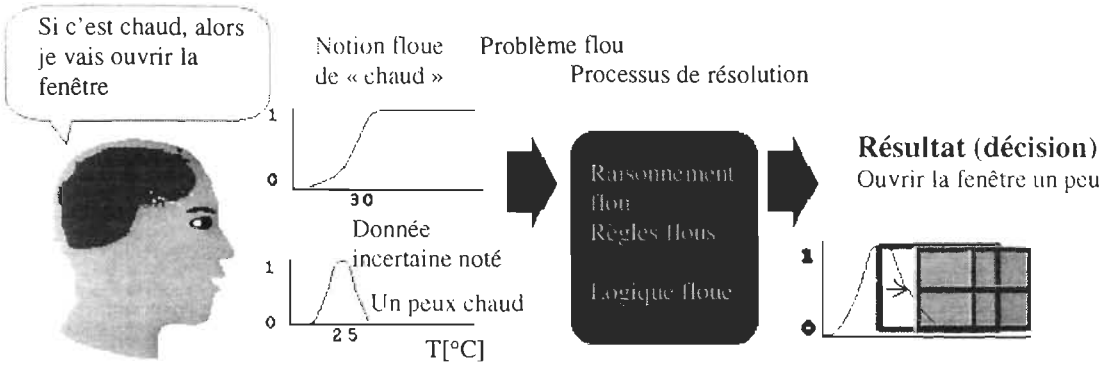


Figure 3.1 : Analogie du raisonnement humain avec la logique floue

➢ *définition d'un ensemble flou*

Soit un ensemble de référence (ou univers) U, on définit un ensemble flou A dans U par la donnée d'une application μ_A de U dans l'intervalle réel [0,1]. À tout élément x ∈ U on associe une valeur $\mu_A(x)$ tel que

$$0 \leq \mu_A(x) \leq 1 \quad (\mu_A : U \rightarrow [0,1]) \qquad (3.1)$$

L'application μ_A est appelée fonction d'appartenance de l'ensemble flou A. À tout élément x de U, la valeur μ_A associée n'est pas nécessairement égal à 0 ou 1, elle est à priori quelconque et désigne le degré d'appartenance de x à l'ensemble A. On peut distinguer trois cas :

- $\mu_A(x) = 0$: C'est-à-dire que x ne satisfait pas du tout la propriété vague sous-entendue par A.

- $\mu_A(x) = 1$: C'est-à-dire x satisfait pleinement la propriété vague définit par A

- $\mu_A(x) \neq 0,1$: On dit que x appartient partiellement à l'ensemble flou A. On peut dire aussi que x ne satisfait que partiellement à un certain degré $\mu_A(x)$ la propriété vague définit par A.

➢ *variables linguistiques & Opérateurs logiques*

La notion de variable linguistique a été introduite par Zadeh ; Elle suggère d'emblée que les valeurs de ces variables ne sont pas numériques, mais plutôt symboliques, en termes de mots ou d'expression du langage naturel. De manière générale on définit une variable linguistique comme appartenant à un intervalle [0,1] et associée à la fonction d'appartenance μ_A.

Les variables linguistiques sont reliées entre elles par des opérateurs logique pour former ce qu'on appelle les règles d'inférence. Lorsque plusieurs variables d'entrée (prémisses) contribuent à faire varier une sortie, l'implication logique utilise les opérateurs booléens classiques «ou » et le « et » par exemple:

Si (température est « fraîche » ou degré hygrométrique est « humide » alors « ouverture du robinet « grand ouverte »)

Dans cette implication, comment trouver le facteur d'appartenance de la sortie (ouverture du robinet) ? Il existe plusieurs façon de faire, la plus répandue semble celle qui a été formulé par Mandani comme suit:

À l'opérateur logique « ou » est associé la fonction MAX.

À l'opérateur logique « et » est associé la fonction MIN.

À l'opérateur logique « non » est associé la fonction complément à 1.

Ainsi, pour deux variables d'entrée x_1 et x_2 qui impliquent une variable de sortie s avec des coefficients d'appartenance respectifs μ_1, μ_2 et δ :

$(x_1 \in E_1 \text{ ou } x_2 \in E_2)$ *implique* $(s \in S_1)$
on aura $\delta = \max(\mu_1, \mu_2)$
$(x_1 \in E_1 \text{ et } x_2 \in E_2)$ *implique* $(s \in S_1)$
on aura $\delta = \min(\mu_1, \mu_2)$

où E_1 et E_2 sont les deux ensembles flous dans l'espace d'entrée et S_1 l'ensemble flou de sortie.

3.1.2 Éléments d'un système à base de logique floue

Tout système à base de logique floue peut être décomposé essentiellement en trois blocs (voir figure 3.2) : fuzzification, inférence et défuzzification.

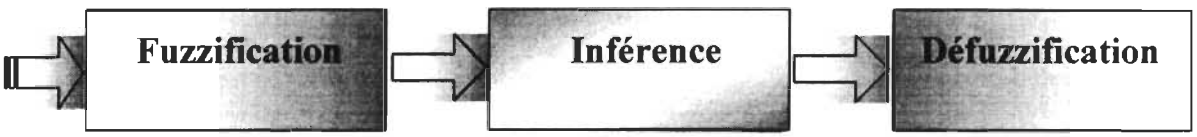


Figure 3.2 : Différents bloques d'un système à base de logique floue

On va définir ces trois blocs par un exemple pratique tiré de [MAU97]. Dans cet exemple nous désirons doser le freinage d'un véhicule (variable de sortie) en fonction de la vitesse et la distance à l'obstacle (variables d'entrée).

- *Fuzzification des variables*

Dans cette partie on définit les variables d'entrée et on caractérise, de manière numérique, l'imprécision qui peut exister sur ces valeurs.

Entrée[2]

Vitesse du véhicule (de 0 à 140 km/h) : *lent* *moyen* *rapide*

Proximité de l'obstacle (de 0 à 100m) : *proche* *éloigné*

Sortie

Freinage (sur une échelle de 0 à 10) : *Faible* *Modéré* *énergique*

On choisit une forme triangulaire (voir figure 3.3) pour les fonctions d'appartenance sauf aux extrémités.

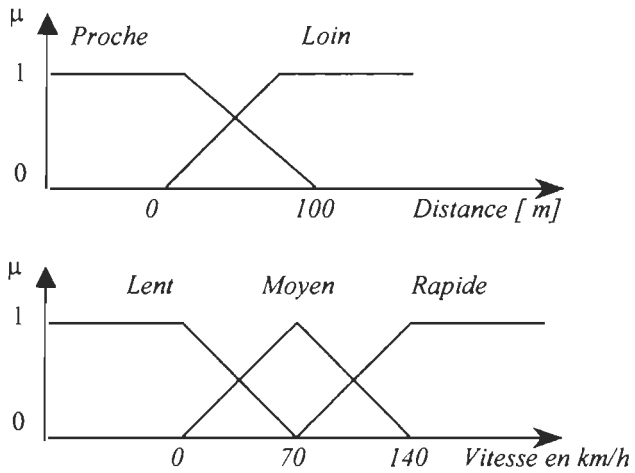


Figure 3.3 : Fonction d'appartenance d'entrée au système

➢ *Formulation des règles d'inférence*

Dans le bloc inférence, les valeurs des variables linguistiques sont liées par plusieurs règles qui doivent tenir compte du comportement statique et dynamique du système à régler ainsi que du but du réglage envisagé. Ces règles sont formulées sous la forme suivante:

Si [(obstacle = = proche) et (vitesse = = moyenne)] alors (freinage =modéré)

Si [(vitesse = = rapide)] alors (freinage = énergique)

[2] « vitesse du véhicule » et « proximité de l'obstacle » sont les variables linguistiques. Lent, moyen, Rapide sont les enbsembles foues.

Si [(obstacle = = loin) et (vitesse = = lente)] alors (freinage =faible)

La figure 3.4 résume l'application des trois règles d'inférences. supposons le cas où les valeurs des variables d'entrée sont:

Distance à l'obstacle 60m ce qui implique que l'obstacle est près à 30% et loin à 70%.

Vitesse du véhicule 56km/h ce qui implique que la vitesse est moyenne à 80% et rapide à 0%.

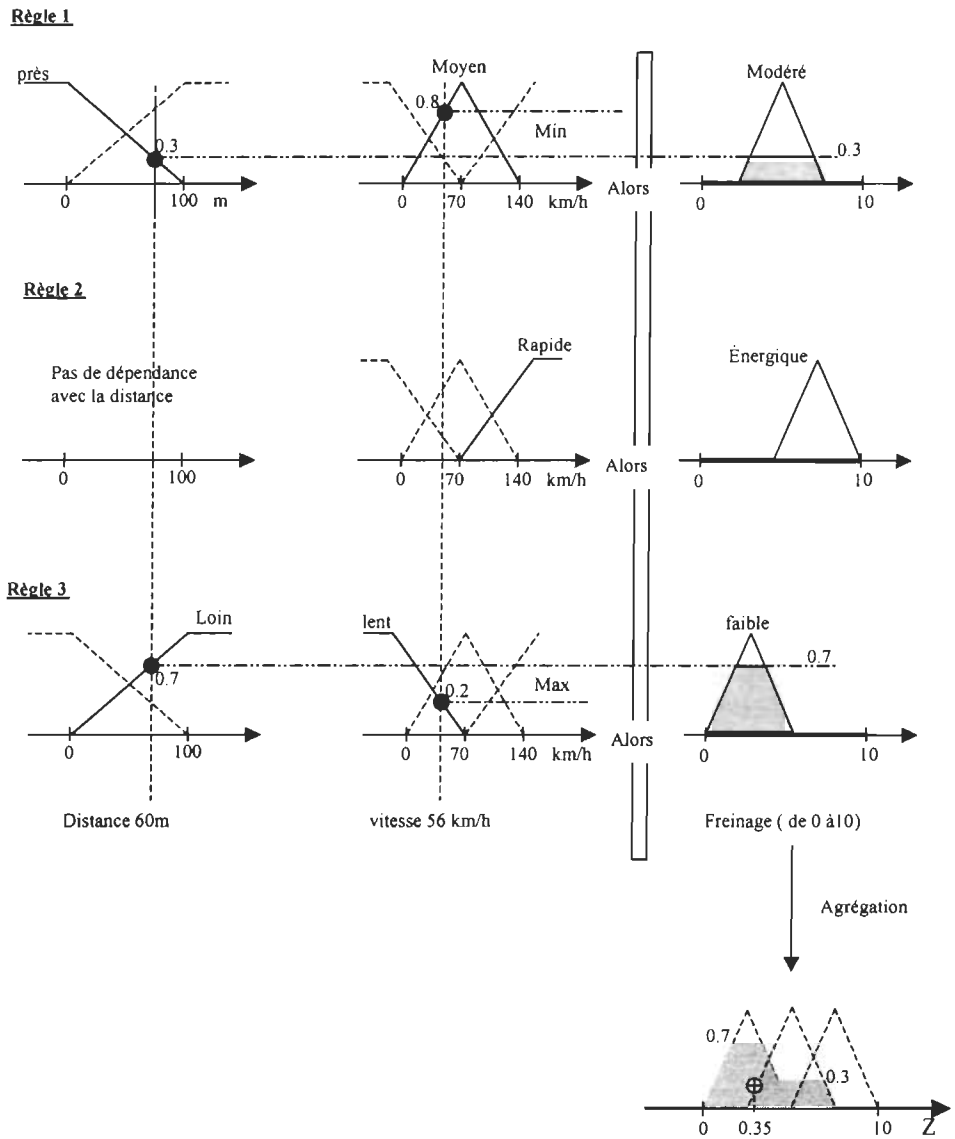


Figure 3.4: Fuzzification – règles d'inférences – défuzzification

Nous observons que plusieurs combinaisons de valeurs des variables d'entrée n'interviennent pas dans les prémisses. Par exemple :

Si [(obstacle = = loin) et (vitesse = = rapide)] n'est pas cité ce qui signifie que l'expert n'a pas d'opinion sur cette situation.

La somme logique (ou) de tous ces ensembles flous conduit, en dernier ressort, à un ensemble flou final dont la forme est la conséquence de l'opération MAX. C'est à partir de cette forme qu'on doit tirer la décision à prendre. Ceci détermine l'opération de défuzzification.

➢ *Défuzzification*

Cette étape permet d'affecter une valeur précise comprise entre 0 et 10 au freinage, déduite de l'ensemble floue de sortie. Il existe plusieurs méthodes :

Méthode COG (The center of gravity of the area)

On cherche le centre de gravité de la surface obtenue. L'abscisse donne la valeur de freinage défuzzifiée. Soit Z l'abscisse de la courbe qui définit la surface de défuzzification (figure 3.4). Les coefficients W_i désignent la valeur correspondante à chaque Z_i le i^{eme} échantillon d'entrée sur un nombre total de n échantillons

$$Z_{out} = \frac{\sum_{i=0}^{n} w_i . z_i}{\sum_{i=0}^{n} w_i} \qquad (3.2)$$

Par la méthode de défuzzification par le centre de gravité on obtient dans notre exemple une valeur de Z_{out}=0.35, Donc on aura un freinage de 3.5 sur une échelle de 0 à 10.

Méthode MOA (The middle of the area)

$$\sum_{i=0}^{h} w_i = \sum_{i=h}^{n} w_i \qquad z_h = z_{out} \tag{3.3}$$

Où $0 < h < n$, h représente l'indice pour lequel l'égalité de l'équation (3.3) est vérifiée et n est le nombre total de coefficients w_i. Avec cette méthode on trouve Z_h=3.5 avec n=20

Méthode MOM

$$z_{out} = \frac{\sum_{i \in M} z_i}{|M|} \qquad M = \arg_i Max\{w_i\} \tag{3.4}$$

3.1.3 Propriété de la logique floue

➢ *Avantage par rapport aux méthodes conventionnelles*

Les méthodes conventionnelles se basent sur une modélisation adéquate du système à corriger mises souvent sous forme de fonction de transfert analytique ou d'équations d'états. Cela nécessite des notions avancées du modèle mathématique du problème. Par contre, la logique floue donne une approche plutôt pragmatique, permettant d'inclure des expériences acquises par des opérateurs.

➢ *Prise en compte des données imprécises ou incertaines*

Il est trés important de ne pas être obligé de seuiller tout de suite les notions d'entrée d'un système de décision, car une erreur de 1% sur la valeur du seuil peut provoquer une erreur de 100% sur la décision. C'est ce que montre la figure 3.5 dans le cas simple ou la décision est justement l'appartenance ou non à l'ensemble défini par ce seuil.

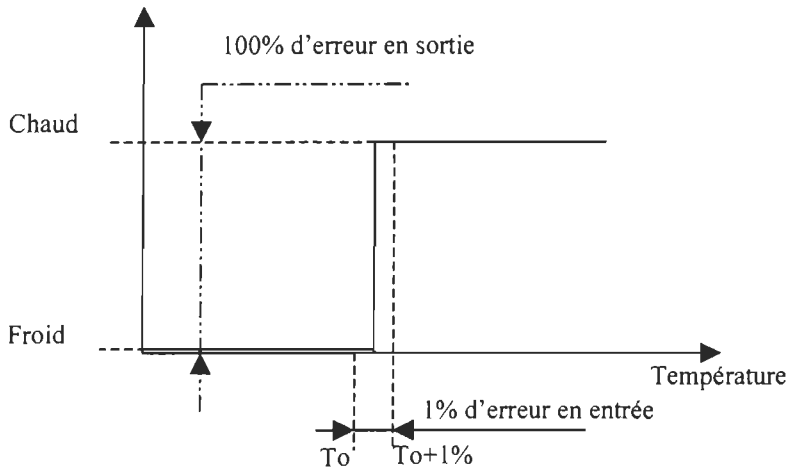


Figure 3.5 : logique classique et données imprécises

La logique floue permet de manipuler des notions sémantiquement contradictoires mais numériquement non exclusives (figure 3.6).

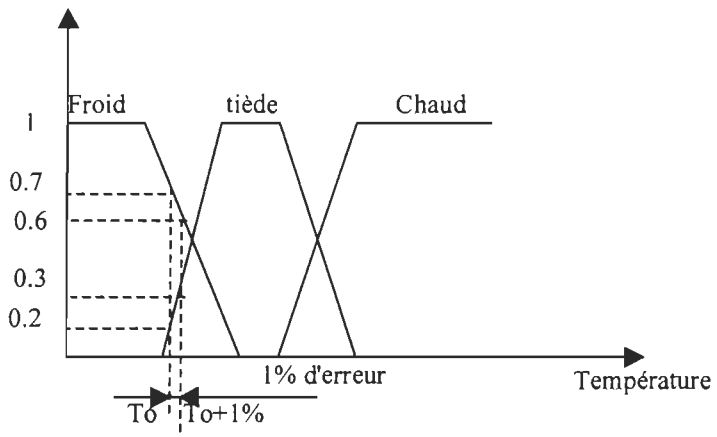


Figure 3.6 : Logique floue et données imprécises

3.1.4 Exemple d'application

L 'application de la logique floue a été trés utilisée par les industries japonaises en grande partie pour la régulation d 'appareils de grande consommation telle que la machine à laver,

le réfrigérateur, la mise au point automatique pour les appareils photos aussi pour les tunneliers, le contrôle du débit de la température de l 'eau, le four de verrerie, le métro, etc.

3.2 Application de la logique floue à l'égalisation de canaux

3.2.1 Principe

Selon le principe énoncé à la section 2.3.2 (figure 2.4), l'idée est d'accorder à chaque *x(k)* échantillon d'entrée à l'instant n un degré d'appartenance à chacune des fonctions d'appartenance qui couvrent l'espace d'entrée. Cette étape va déterminer le bloc de fuzzification. La seconde étape consiste à utiliser un algorithme adaptatif qui permet de construire les règles d'inférence en optimisant l'erreur entre la sortie du canal et le signal idéal. La dernière étape consiste à extraire la décision à prendre à partir de ces règles ce qui va constituer le bloque de défuzzification. La sortie de ce bloc va fournir l'estimé $\hat{s}(k)$. En se basant sur ce principe de l'égaliseur [WAN93] on va développer le fonctionnement de l'algorithme proposé et faire une étude approfondie qui permettra de le simplifier pour une implantation ultérieure.

3.2.2 Réalisation de l'égaliseur à base de logique floue

Considérons le vecteur $\tilde{\mathbf{r}}(k)$de valeurs réelles et une séquence de valeurs réelles *s(k)*, Avec k=1, 2 … , K est l'index de temps.

$$\tilde{r}(k) \in U = \left[C_1^-, C_1^+\right] \times \left[C_2^-, C_2^+\right] \times \ldots\ldots\ldots \times \left[C_n^-, C_n^+\right] \subset R^n \tag{3.5}$$

Où $\left[C_i^-, C_i^+\right]$ représente les intervalles des ensembles flous d'entrée et k le nombre total d'intervalles. À chaque instant n on a les valeurs $\tilde{r}(k)$ et *s(k)*, le problème est de déterminer le filtre adaptatif à chaque instant n. C'est ici où intervient le filtre RLS pour minimiser le

critère J(k). La fonction f_k fait correspondre aux valeurs de l'ensemble d'entrée une valeur réelle $\hat{x}(k)$ qui correspond à la sortie de l'égaliseur.

$$f_k : U \subset R^n \to R$$

$$J(k) = \sum_{i=0}^{k} \lambda^{k-i} \left[s(i) - f_k(\tilde{r}(i)) \right]^2 \qquad (3.6)$$

λ est dit le facteur d'oubli.

Comme on a vu dans le paragraphe précédent, il faut définir des fonctions d'appartenance pour réaliser le bloc de fuzzification. ces fonctions sont définies par $\mu_{F_i^j}(X_i)$ pour chaque intervalle $\left[C_i^-, C_i^+\right]$. En se referant à [WAN93], les équations des fonctions d'appartenance sont définies comme suit :

$$\mu_{F_i^j}(x_i) = \exp\left[-\frac{1}{2} \left(\frac{x_i - \overline{x_i^j}}{0.3} \right)^2 \right] \qquad (3.7)$$

$\overline{x}_i^j$ définit la valeur où la fonction d'appartenance $\mu_{F_i^j}$ attend sa valeur maximale.

$$x_i = \tilde{r}(k)$$

La deuxième étape est la réalisation du bloc d'inférence qui est formé de règles :

$$R^{(j1 \ldots jn)} : Si \;\; X_1 \in F_1^{j1} \;\; et \ldots et \; X_n \in F_n^{jn} \quad Alors \quad d \in G^{(j1 \ldots jn)}$$

Ces règles seront déterminées par un filtre adaptatif tel que RLS ou LMS qui permet de les construire en minimisant à chaque itération l'erreur e(n) définit comme suit.

$$e(k) = s(k) - \hat{x}(k) \qquad (3.8)$$

Dans notre cas ces règles sont représentées par le vecteur θ définit par l'équation (3.9). Ce dernier est une concaténation de m_2 vecteurs dont chacun est composé de m_1 éléments [WAN93].

$$\theta = \left[\theta^{(1,1,\ldots,1)}(x) \ldots\ldots\ldots\ldots \theta^{(m_1,1,\ldots,1)}(x)\right. \\ \theta^{(1,2,\ldots,1)}(x) \ldots\ldots\ldots\ldots \theta^{(m_1,2,\ldots,1)}(x) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \left.\theta^{(1,m_2,\ldots,m_n)}(x) \ldots\ldots\ldots\ldots \theta^{(m_1,m_2,\ldots,m_n)}(x)\right] \qquad (3.9)$$

Les valeurs du vecteur θ sont adaptées par l'algorithme LMS, Eq. (3.10), ou l'algorithme RLS, Es. (3.11), résumé dans le tableau 3.1.

Tableau 3.1 : Algorithme LMS et RLS pour l'adaptation de θ

LMS :	
$\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) + \mu.e(k).P$ $e(k) = S(k) - \hat{x}(k)$	(3.10)
RLS	
$\boldsymbol{\varphi}(k) = \mathbf{p}(x(k))$ $\mathbf{P}(k) = \frac{1}{\lambda}\left[\mathbf{P}(k-1) - \mathbf{P}(k-1)\boldsymbol{\varphi}(k)\left(\lambda + \boldsymbol{\varphi}^T(k)\mathbf{P}(k-1)\boldsymbol{\varphi}(k)\right)^{-1}\boldsymbol{\varphi}^T(k)\mathbf{P}(k-1)\right]$ $\mathbf{K}(k) = \mathbf{P}(k-1)\boldsymbol{\varphi}(k)\left[\lambda + \boldsymbol{\varphi}^T(k)\mathbf{P}(k-1)\boldsymbol{\varphi}(k)\right]^{-1}$ $\boldsymbol{\theta}(k) = \boldsymbol{\theta}(k-1) + \mathbf{K}(n)\left(S(k) - \boldsymbol{\varphi}^T(k)\boldsymbol{\theta}(k-1)\right)$	(3.11)

La dernière étape est d'extraire la décision à prendre de ces règles ce qui constitue l'étape de défuzzification. Cette étape peut être réalisée par plusieurs méthodes. Dans notre cas on utilise la méthode de centroïde [WAN92]. Le résultat de la défuzzification est donné par l'équation (3.12):

$$\hat{x}_k(\tilde{r}) = \frac{\sum_{j1=1}^{m_1} \cdots \sum_{jk=1}^{m_k} \theta^{(j1 \cdots jk)} . (\mu_{F_1^{j1}}(\tilde{r}_1) \cdots \mu_{F_i^{jk}}(\tilde{r}_k))}{\sum_{j1=1}^{m_1} \cdots \sum_{jk=1}^{m_k} (\mu_{F_1^{j1}}(\tilde{r}_1) \cdots \mu_{F_n^{jk}}(\tilde{r}_k))} \quad (3.12)$$

$$p^{(j1 \cdots jk)}(\tilde{r}) = \frac{\mu_{F_i^{j1}}(\tilde{r}_1) \cdots \mu_{F_k^{ijk}}(\tilde{r}_k).}{\sum_{j1=1}^{m1} \cdots \sum_{jk=1}^{mk} (\mu_{F_i^{j1}}(\tilde{r}_1) \cdots \mu_{F_n^{ijk}}(\tilde{r}_k))} \quad (3.13)$$

$p^{(j1,\ldots,jn)}(\tilde{r})$ représente les fonctions floues qui font associer à chaque $\tilde{r}$, valeur d'entrée à l'égaliseur une valeur qui sera déterminée en fonction de son appartenance aux différents ensembles flous donnés par la fonction d'appartenance $\mu_{F_i^{ij}}$. Il s'écrit de la manière suivante :

$$\begin{aligned} p = \big[& p^{(1,1,\ldots,1)}(\tilde{r}) \cdots p^{(m_1,1,\ldots,1)}(\tilde{r}) \\ & p^{(1,2,\ldots,1)}(\tilde{r}) \cdots p^{(m_1,2,\ldots,1)}(\tilde{r}) \\ & \cdots \\ & p^{(1,m_2,\ldots,m_k)}(\tilde{r}) \cdots p^{(m_1,m_2,\ldots,m_k)}(\tilde{r}) \big] \end{aligned} \quad (3.14)$$

En se basant sur (3.12) et (3.13) on peut écrire

$$\hat{x}_k(\tilde{r}) = p^T(\tilde{r}).\theta \quad (3.15)$$

Enfin l'égaliseur ainsi formulé peut être représenté d'une manière schématique la figure3.7.

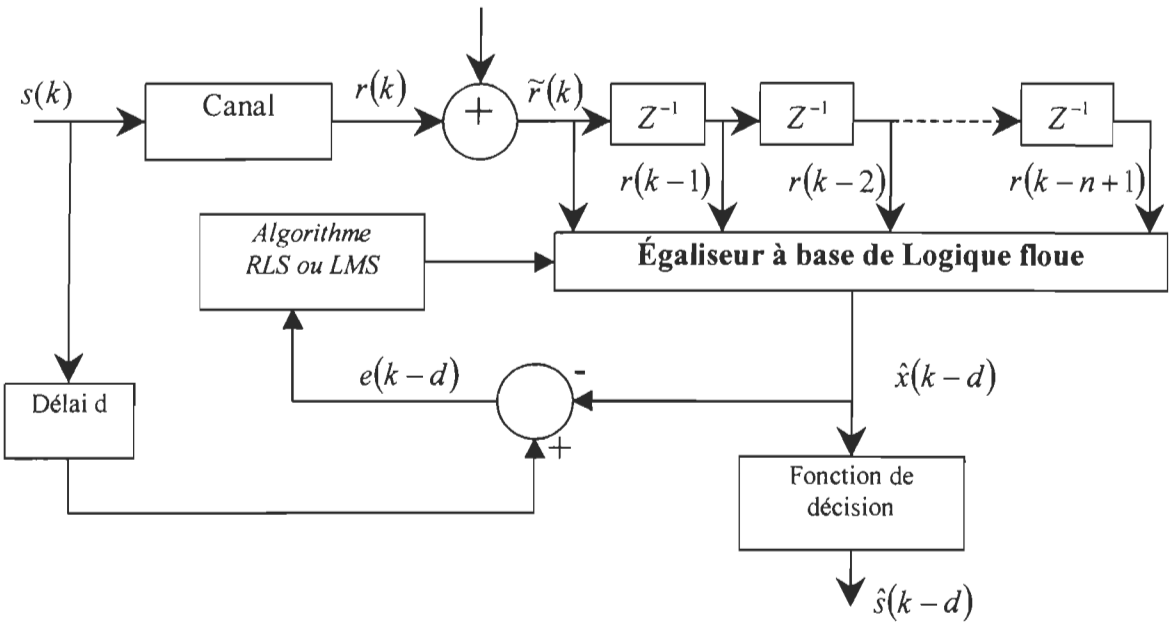


Figure 3.7 Égaliseur de canaux à base de logique floue

3.4 Simulation dans l'environnement Matlab®

Dans cette section on aura à valider le fonctionnement de l'algorithme d'égalisation de canaux à base de logique floue formulé au paragraphe précédent. Par conséquent, on a à comparer ces performances par rapport à d'autres algorithmes tels que LMS et RLS. Les programmes Matlab sont disponibles à l'annexe A. En outre on présentera sa robustesse au bruit et à la non-linéarité du canal. La base de la comparaison est le BER « Bit Error Rate » qui est défini en terme de pourcentage par l'équation (3.16).

$$BER = \frac{Nombre \text{ de bits erronnés}}{Nombre \text{ de bits transmis}} \times 100 \qquad (3.16)$$

Le BER est le quotient du nombre de bits erronés par le nombre total de bits transmis. C'est un indicateur trés utile pour évaluer la vitesse de convergence des algorithmes et leur robustesse au bruit. Il est le plus souvent représenté sur une échelle logarithmique.

3.3.1 Canaux utilisés

On va utiliser deux types de canaux pour la simulation et la comparaison des différents algorithmes :

➢ *Canal linéaire selon [HAY96]*

$$h_k = \begin{cases} \frac{1}{2}\left[1+\cos\left(\frac{2\pi}{W}(k-2)\right)\right], \text{n} = 1,2,3 \\ 0 \text{ sinon} \end{cases} \qquad (3.17)$$

C'est un canal dont la réponse impulsionnelle h_n est à trois points, W est le paramètre qui définit la largeur du canal.

➢ *Canal non-linéaire selon [WAN93],[VID99]*

On va choisir deux types de canaux non linéaires le premier représenté par l'équation (3.18), proposé par [WAN93] et le deuxième par l'équation (3.19), est fortement non linéaire [VID99].

Canal 1 :

$$\tilde{y}(k) = s(k) + 0.5s(k-1) - 0.9\left[s(k) + 0.5s(k-1)\right]^3 + \eta(k) \qquad (3.18)$$

Canal 2 :

$$y_n = 0.5x + x^2 + x^3$$
$$où\ x = 0.25s(k-2) + s(k-1) + 0.25s(k) \qquad (3.19)$$

3.3.2 Influence du nombre de fonctions d'appartenance

Pour choisir le nombre de fonction d'appartenance ("Membership Function", MF) au niveau du bloc de fuzzification on va comparer les résultats de simulation pour MF=3, 5 et 7 pour le canal linéaire de l'équation (3.17) et non linéaire de l'équation(3.19). La figure 3.8 indique la variation du BER en fonction du SNR « Signal Noise Rate » utilisant LMS

pour l'adaptation des paramètres. Le nombre d'échantillons k, pour l'adaptation est fixé à 500 avec μ=0.2. La valeur de μ influe sur la vitesse de convergence de LMS vers un minimum d'erreur. Une valeur trés élevée peut conduire le système à diverger. Donc on a choisi d'avoir une valeur moyenne de μ pour assurer une convergence rapide.

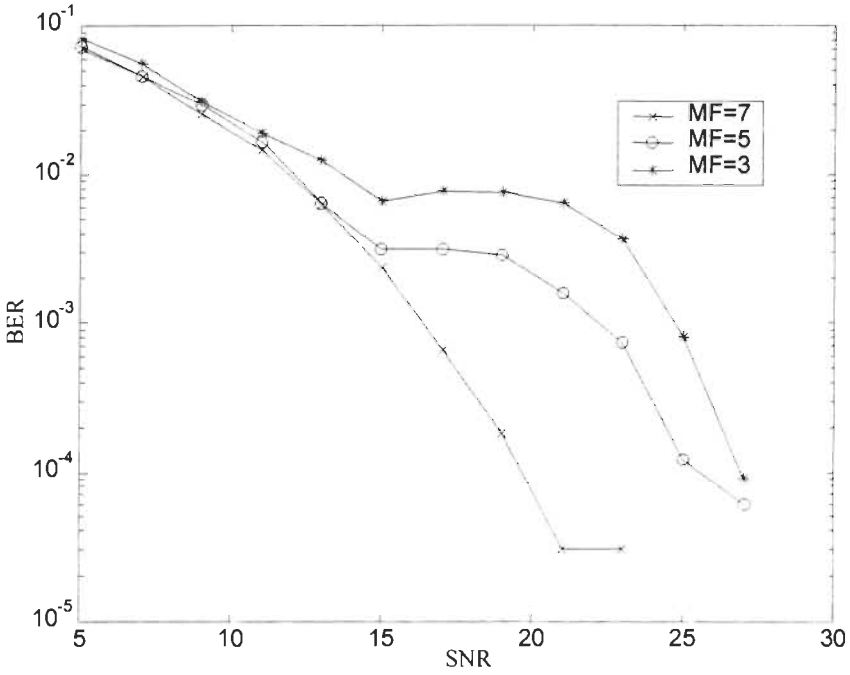


Figure 3.8 : Influence du nombre de fonction d'appartenance (en utilisant un canal linéaire)

On remarque que pour un canal linéaire, la qualité de l'égalisation s'améliore en fonction du nombre de fonctions d'appartenance. Le BER devient acceptable pour $MF \geq 7$.

Les mêmes simulations ont été réalisées pour le canal non-linéaire donné par l'équation (3.19). la figure 3.9 résume les résultats obtenus.

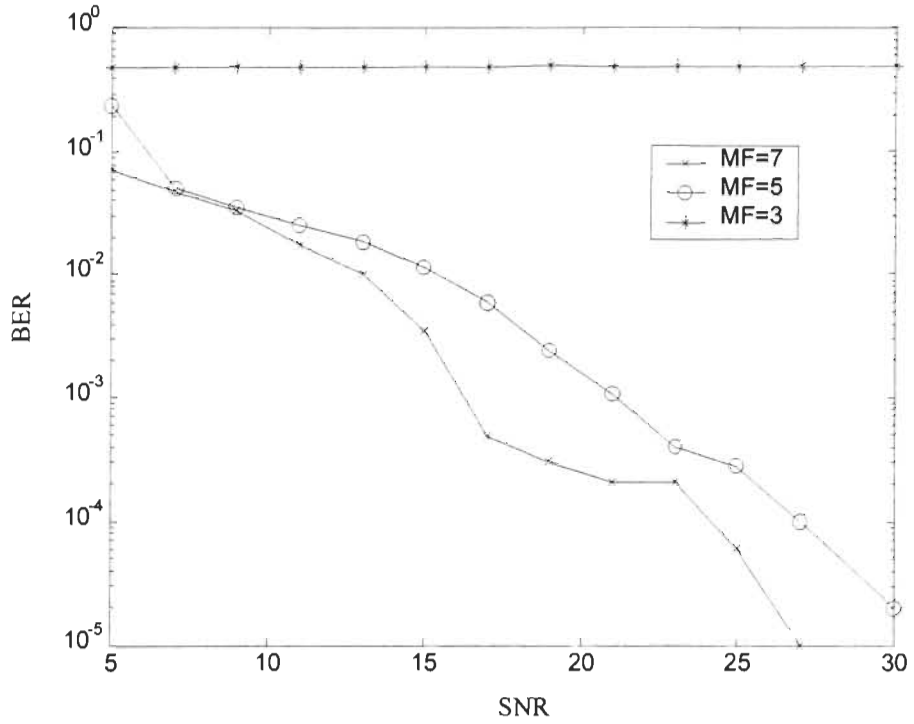


Figure 3.9 : Influence du nombre de fonction d'appartenance (en utilisant un canal non-linéaire)

On remarque que pour un canal non-linéaire trois fonctions d'appartenance dans le bloc de fuzzification n'est pas suffisant pour éliminer l'effet du canal.

3.3.3 Comparaison de la logique floue avec LMS et RLS.

Pour justifier l'utilisation de l'algorithme d'égalisation de canaux non-linéaires à base de logique floue, on doit le comparer à d'autres méthodes. La figure 3.10 permet de le comparer l'algorithme basé sur la logique floue avec adaptation des paramètres par LMS (LF_LMS), avec les filtres transverses linéaires RLS et LMS (LT_LMS, LT_RLS) pour le canal linéaire définit par l'équation 3.17. La figure (3.11) donne le résultat de simulation pour le canal non-linéaire donné par l'équation (3.19).

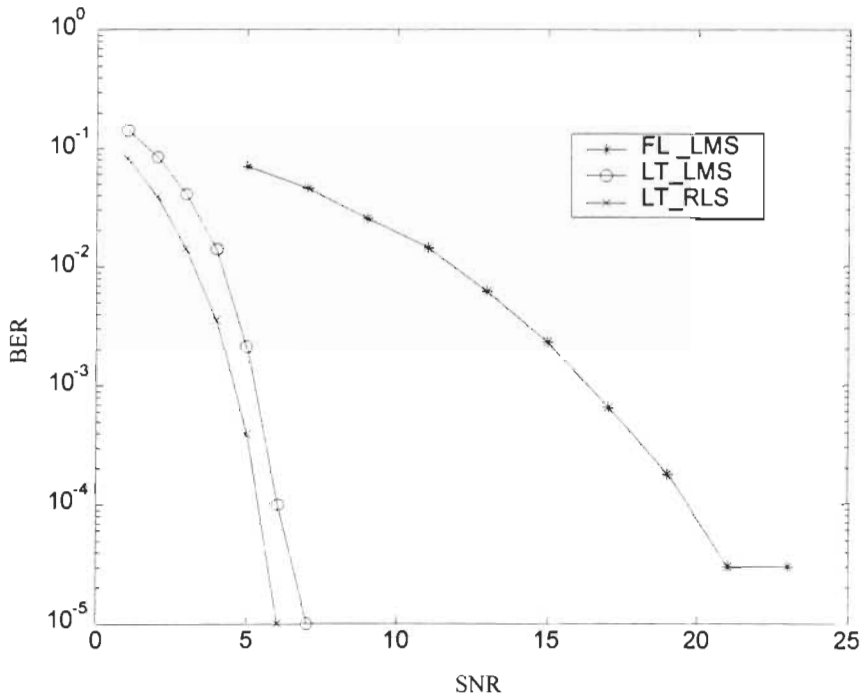


Figure 3.10 : comparaison de la logique floue avec RLS et LMS pour un canal linéaire

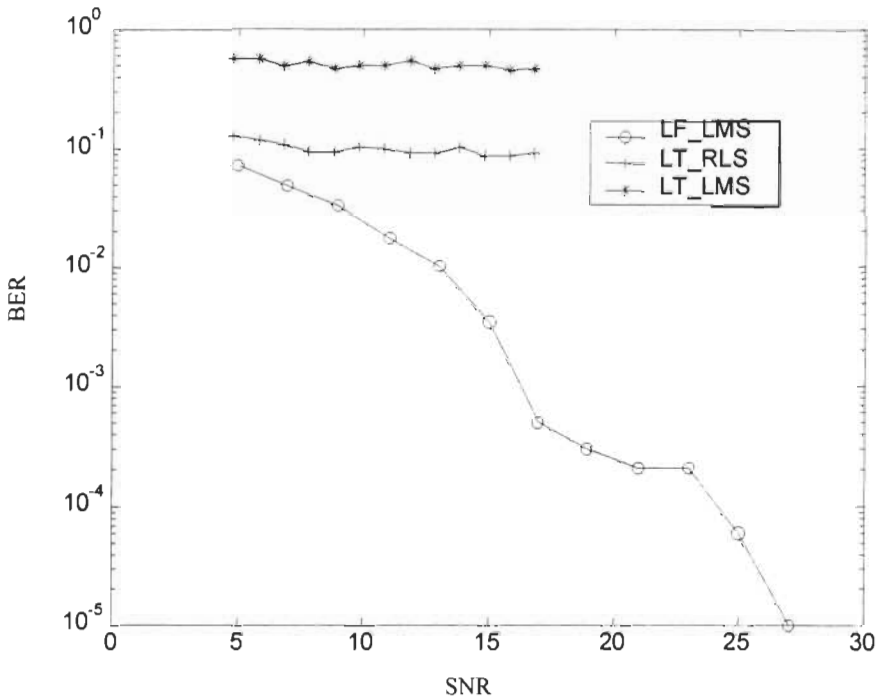


Figure 3.11 : comparaison de la logique floue avec RLS et LMS pour un canal non-linéaire

On remarque que l'algorithme LT_RLS, LT_LMS et l'algorithme à base logique floue (LF_LMS) égalisent pour le cas d'un canal linéaire (figure 3.10). L'algorithme LF_LMS donne des résultats de correction moins bons que LT_RLS et LT_LMS car il demande beaucoup plus d'échantillons pour atteindre son erreur minimale. Par contre la différence entre les trois algorithmes est claire pour un canal non-linéaire. LT_RLS et LT_LMS n'arrivent pas à corriger le canal alors que l'algorithme LF_LMS y arrive (figure 3.11).

3.3.4 Exemples de résultats de simulation

On a fait les simulations dans l'environnement Matlab avec le canal non-linéaire de l'équation (3.18) et différents nombre de fonctions d'appartenance MF=5 et 7 décrits par l'équation (3.7). Les figures 3.12, à 3.14 représentent les résultats de simulation pour un signal de 300 échantillons dont les 70 premiers servant à l'adaptation.

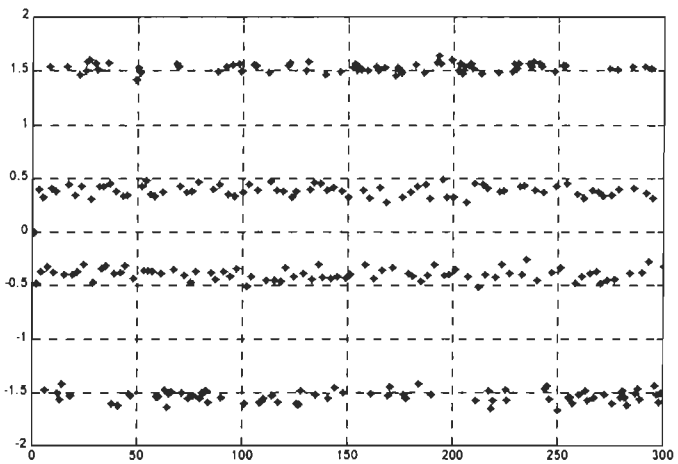


Figure 3.12 : signal de sortie du canal avec SNR=20dB (BER=50%)

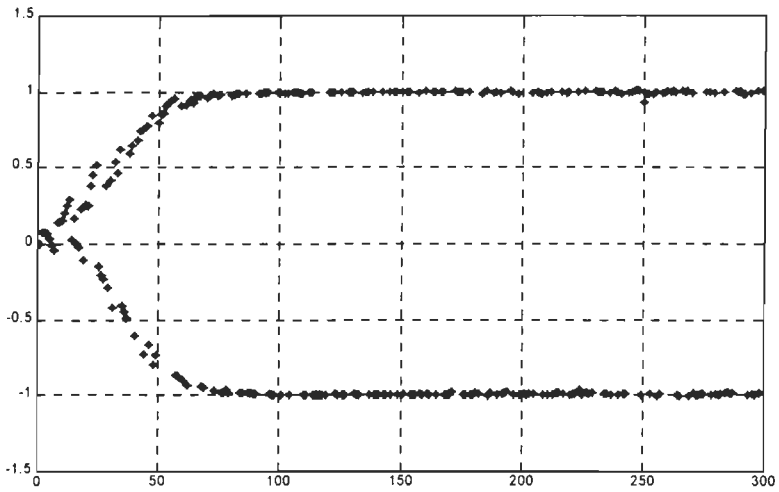


Figure 3.13 : Signal reconstitué avec MF=7 (BER = 1%)

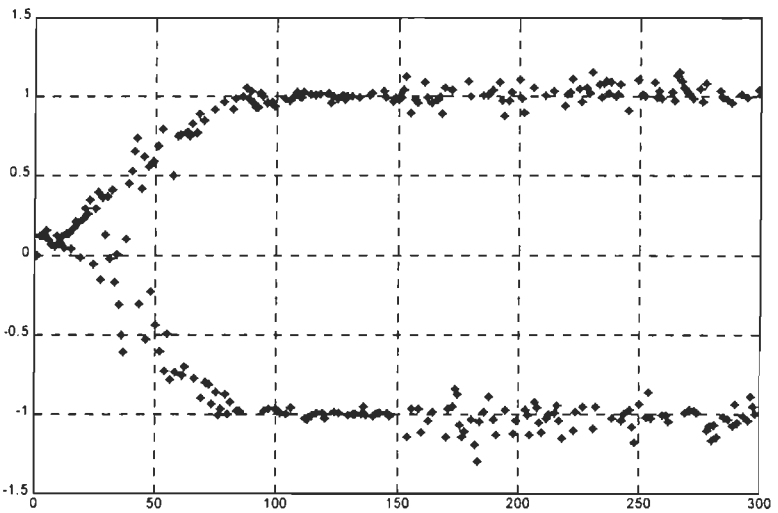


Figure 3.14 : Signal reconstitué avec MF=5 (BER =1.6%)

D'après ces figures on peut remarquer l'influence du nombre des fonctions d'appartenances sur le résultat de l'égaliseur. Lorsque l'on change le nombre des fonctions d'appartenance de 5 à 7, on améliore le BER de 1.6% à 1% .

3.5 Égaliseur en vue d'une implantation ITGE

Pour l'implantation de l'égaliseur en technologie numérique ITGE, il est trés important de diminuer au maximum la complexité de calcul. Le but est d'arriver avec une architecture offrant le maximum de débit, avec une surface d'intégration raisonnable. L'égaliseur tel qu'il est modélisé à la section 3.2 présente des contraintes importantes quant-à une implantation en ITGE. La première contrainte est l'utilisation des fonctions d'appartenance de type gaussienne dans l'étage de fuzzification. La seconde est la complexité de calcul que présente l'utilisation de l'algorithme RLS pour l'adaptation des paramètres du modèle de logique floue. Enfin la plus importante des contraintes est la dimension des vecteurs manipulés θ et p. L'objectif de cette partie est l'étude de l'algorithme afin d'introduire des simplifications, et permettre une meilleure implantation.

3.5.1 Contraintes d'implantation

Les contraintes reliées à l'implantation en technologie VLSI sont les suivantes:

- Implantation des fonctions d 'appartenance en Gaussienne
- Implantation de l'opération de division
- Dimensions des vecteurs manipulés
- Utilisation de RLS pour l 'adaptation des paramètres

Dans le but de franchir tous ces obstacles, on va commencer par voir l'influence de remplacement des fonctions gaussienne par les fonctions CLM (canoniques linéaires par morceaux, "piecewise"). Ensuite on étudiera l'influence du nombre de fonctions d'appartenance sur la qualité de reconstitution. Enfin pour diminuer la complexité de

calcul, on étudiera l'effet du remplacement de LT_RLS par LT_LMS pour l'adaptation des coefficients formant le vecteur θ.

3.5.2 Utilisation des fonctions canonique linéaire par morceaux

Dans ce paragraphe on va voir l'effet de l'utilisation des fonctions CLM (piecewise) définies par l'équation (3.20), (figure 3.15b), au lieu des fonctions gaussienne définies par l'équation (3.7), (figure 3.15a). La comparaison des résultats sera faite en se basant sur le taux de BER du signal de sortie.

$$\mu(\tilde{y}_{n-i+1}) = \begin{cases} 1 - \dfrac{\left|\tilde{y}_{n-i+1} - \bar{y}_{n-i+1}^{l_j}\right|}{\sigma_i^l} & for \left|\tilde{y}_{n-i+1} - \bar{y}_{n-i+1}^{l_j}\right| < \sigma_i^l \\ 0 & \text{sinon} \end{cases} \qquad (3.20)$$

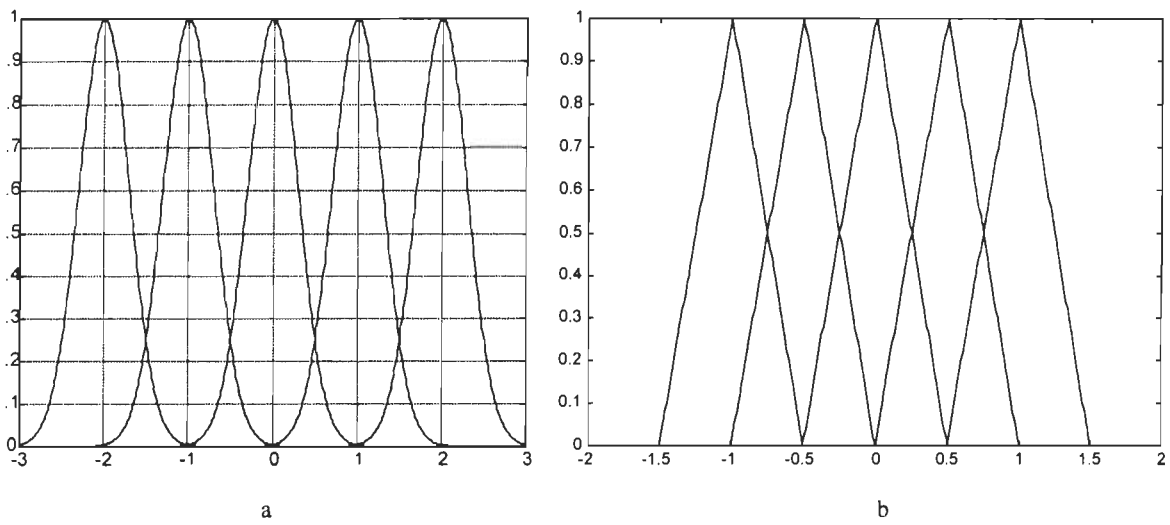


Figure 3.15 : Forme des fonctions d'appartenance a) gaussienne b) canonique linéaire par morceaux (CLM)

Pour la validation de l'utilisation des fonctions CLM on a fait les essais de reconstitution avec le canal non linéaire présenté sur l'équation (3.17), pour une moyenne de 20 itérations

chacune de 1000 échantillons. La figure 3.18 présente la variation du BER pour l'utilisation des fonctions d'appartenance gaussienne et CLM. Le signal de sortie du canal corrompu avec le bruit η est décrit par l'équation (3.21).

$$\tilde{r}(k) = r(k) + \eta(k) \tag{3.21}$$

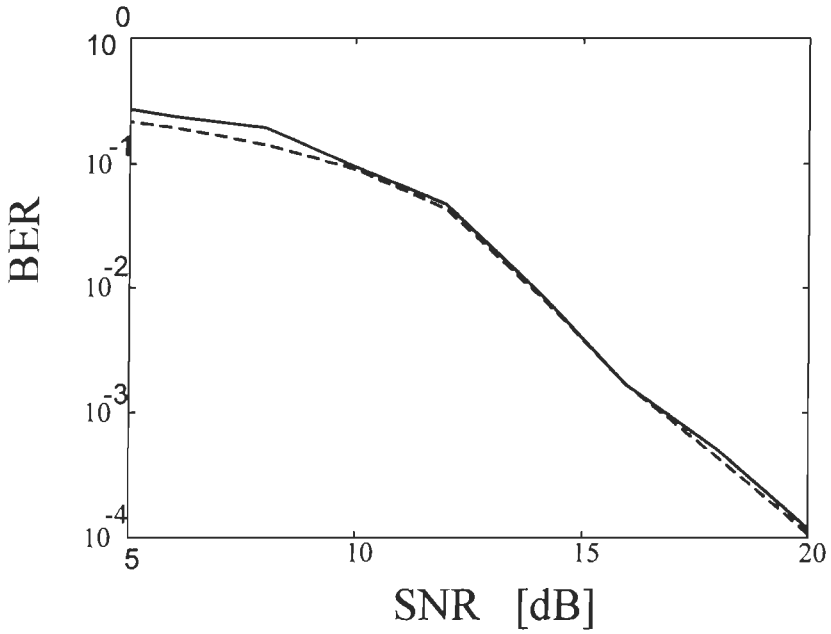


Figure 3.16 : Comparaison du BER pour l'égalisation d'un canal non-linéaire

Ligne continue : Fonction d'appartenance CLM; Ligne interrompue : Fonction d'appartenance gaussienne; 1000 échantillons, moyenne de 15 essais

On remarque que les deux courbes sont presque confondues, et ceci pour un même nombre de fonction d'appartenance (7 fonctions dans ce cas). Cette simulation nous permet de valider l'utilisation de la fonction CLM.

3.5.3 Simplification du calcul

L'étude approfondie de l 'algorithme nous révèle la présence de plusieurs termes nuls au niveau du bloc de fuzzification. Ces termes peuvent être éliminés pour alléger le calcul et économiser le nombre de processeurs pour une implantation ITGE. En effet on remarque que le nombre de termes non nuls du vecteur p (section 3.2) dépend du degré de chevauchement qui existe entre les fonctions d'appartenance $\mu_{F_j^{ij}}$. Supposons qu'on a un degré de chevauchement dc=1 comme le montre la figure (3.17). le nombre de $\mu_{F_j^{ij}} \neq 0$ est deux. D'où un vecteur p avec 4 éléments non nuls.

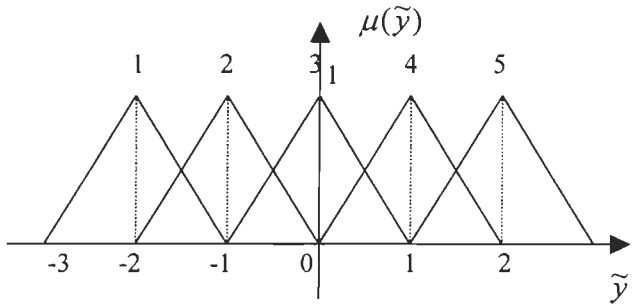


Figure 3.17 : Fonction d'appartenance CLM avec dc=1

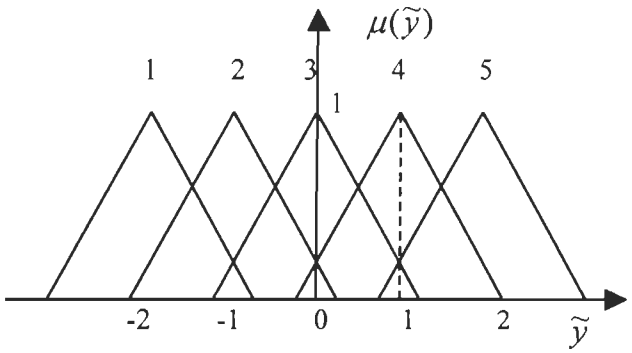


Figure 3.18 : Fonction d'appartenance CLM avec dc=2

Pour dc=1 on a :

Si $\tilde{y}_k = 1.5$ *on aura :* $\mu(\tilde{y}_k, 4) = 0.5$ *et* $\mu(\tilde{y}_k, 5) = 0.5$

Donc pour chaque valeur d'entrée on a au maximum 2 valeurs non nulles.

Pour dc=2 (figure 3.18) on a :

Si $\tilde{y}_k = 1$ *on aura :* $\mu(\tilde{y}_k,3) = 0.25$, $\mu(\tilde{y}_k,4) = 1$ *et* $\mu(\tilde{y}_k,4) = 0.25$

Dans ce cas on a trois valeurs non-nulles, Si on généralise cette approche on aura :

Nombre de valeurs de μ différent de zéro est égale à dc+1.

La simulation sur Matlab avec dc=1 et dc=2 n'a pas donné une grande différence au niveau du BER. Donc pour l'implantation ultérieure on aura intérêt à utiliser dc=1. Cette simplification est trés bénéfique car elle permet de réduire la complexité de calcul. Elle permet aussi de diminuer la dimension des vecteurs manipulés, par conséquent le nombre de processeurs élémentaires utilisés.

3.5.4 Utilisation de LMS pour l'adaptation

D'après les équations (3.10) et (3.11) nous constatons bien que la complexité du calcul de LMS est trés faible par rapport à celle de RLS. En contre parti l'adaptation par LMS nécessite beaucoup plus de cycles. La figure 3.19 représente la comparaison des vitesses de convergence des deux méthodes utilisant 80 itérations pour l'adaptation de θ. Le SNR a été fixé à 13 dB et μ=0.2 pour LMS. Pour RLS on a pris λ=0.92. Cette valeur est choisie après une simulation faisant varier la valeur de λ de 0.82 à 1, sous un SNR de 13 dB utilisant la moyenne de 50000 échantillons (voir figure 3.19). La figure 3.20 montre que la convergence de LMS est plus lente que RLS. Par conséquent l'algorithme LMS est plus favorable pour une implantation ITGE.

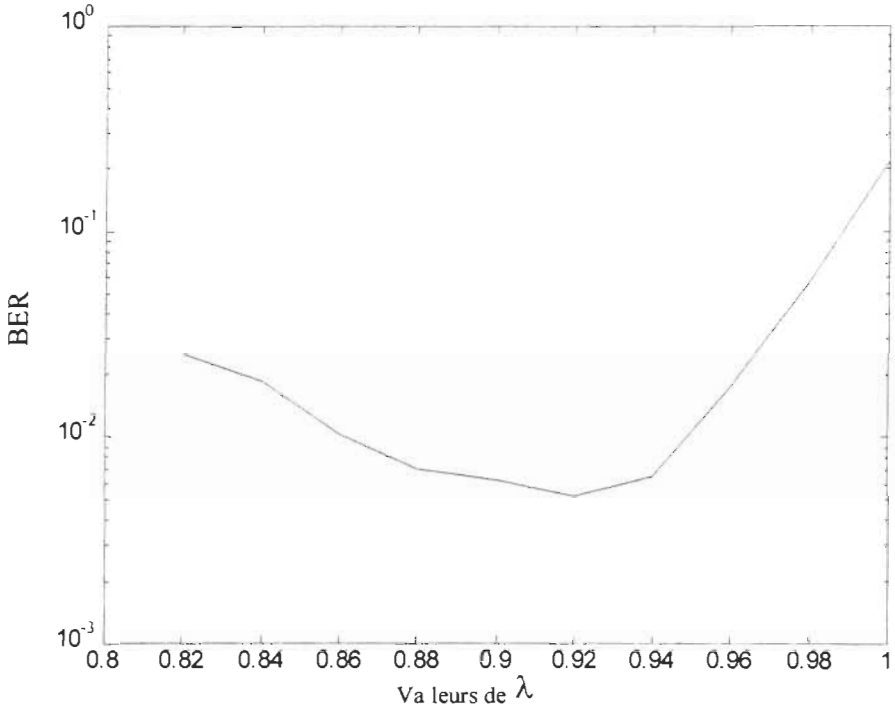


Figure 3.19 : Optimisation de λ

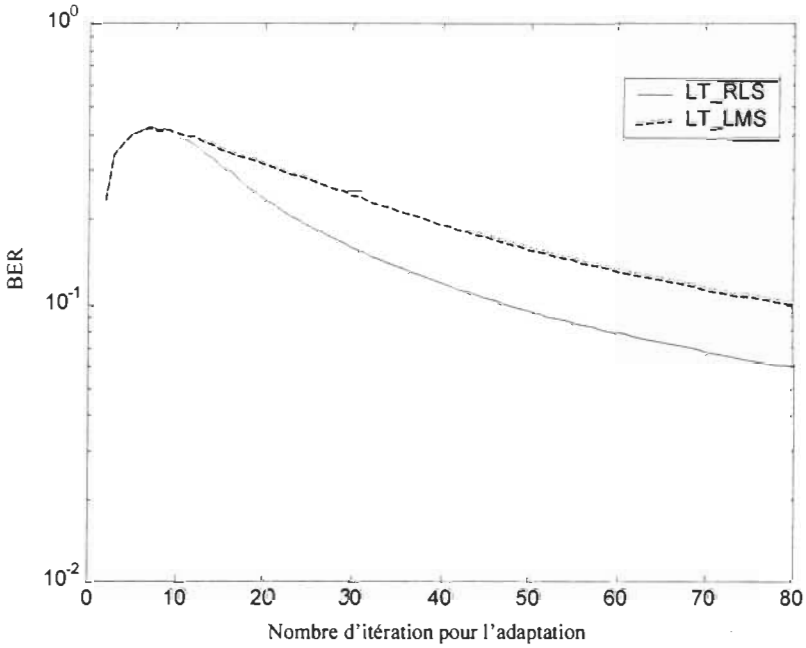


Figure 3.20 : Vitesse de convergence de RLS et LMS

4.6 Conclusion

En conclusion on peut dire que l'algorithme LMS prend plus d'itérations pour donner les mêmes performances que RLS. Par contre l'utilisation de LMS permet de simplifier énormément la quantité de calcul. Par suite LMS est plus convenable pour une implantation ITGE. Le paragraphe 5.4.3 nous permet de simplifier plus la quantité de calcul et ceci en utilisant un degré de chauvauchement de 1. Cette dernière simplification permet de réduire le nombre d'élément du vecteur p à 4 peu importe le nombre de fonctions d'appartenance utilisées. Avec toute ces simplifications on est prêt à procéder à la modernisation de l'architecture de notre système. C'est ce qui va être détaillé au niveau du chapitre 4.

Chapitre 4

Implantation en technologie ITGE

La performance des ordinateurs et des circuits intégrés a connu une amélioration significative pendant les dernières décennies. Pendant longtemps l'augmentation de la vitesse des circuits était telle qu'en suivant l'évolution des composants, il suffit au constructeur de conserver l'architecture séquentielle des machines (figure 4.1) pour pouvoir mettre sur le marché des ordinateurs de plus en plus puissants. L'introduction du pipeline dans les structures à concevoir a donné un nouveau souffle. En effet, sans modification significative de la structure du programme, le pipeline permet un gain important des performances. Les techniques de vectorisation permettent de tirer parti de l'organisation en pipeline du calcul des opérations arithmétiques élémentaires. On peut pipeliner le décodage des instructions, les transferts de mémoire, et aussi les additions et les multiplications.

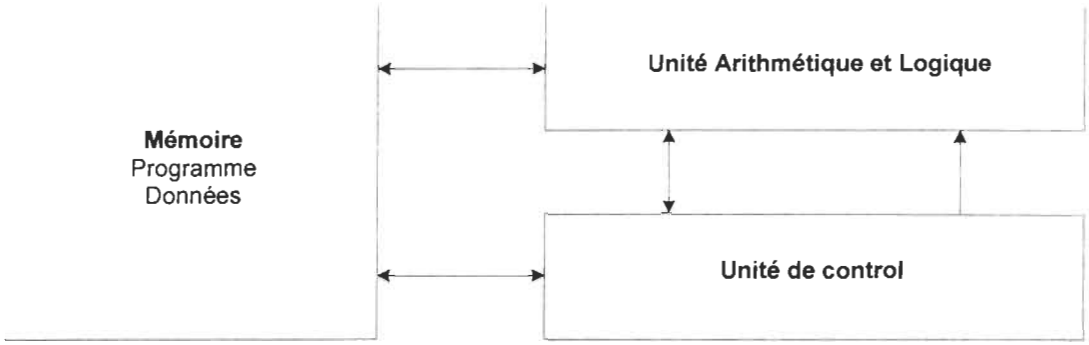


Figure 4. 1 : Le modèle séquentiel

Cependant, l'augmentation des performances n'est plus suffisante pour répondre aux exigences des utilisateurs. En plus l'industrie des semi-conducteurs est plus à même de développer des technologies à trés haute densité d'intégration que de réaliser des circuits ultra-rapide. On atteint les limites physiques qui semblent incontournables. La nouvelle génération de circuits intégrés, trés rapide est celle utilisant l'arséniure de gallium (AsGa). Le prix à payer pour gagner un ordre de grandeur sur la vitesse des composants en utilisant ces nouvelles technologies sera trés vraisemblablement prohibitif pour la plupart des applications. La meilleure solution apportée par les constructeurs de super-ordinateurs est de faire du parallélisme. Ce concept rompt avec l'approche classique qui consiste à gagner de la vitesse en effectuant plus rapidement chaque opération. En calcul parallèle le gain de vitesse provient de la réalisation simultanée de plusieurs opérations. D'où l'apparition de plusieurs types d'architectures parallèles entre autres :

l'architecture SIMD [COS93] qui contient plusieurs unités de traitement supervisées par une unité de contrôle (Figure 4. 2). Toutes les unités de contrôle reçoivent la même instruction (même programme) diffusée par l'unité de contrôle.

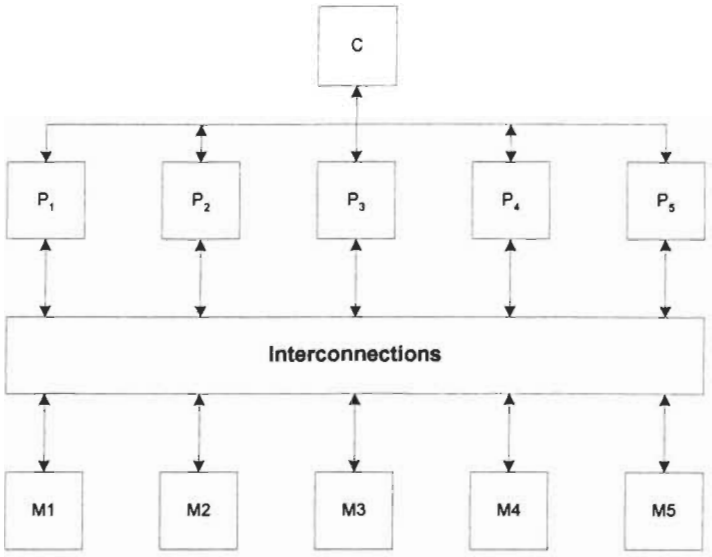


Figure 4. 2 : Architecture SIMD [COS93]

L'architecture MIMD (Figure 4. 3), elle diffère du modèle précédent par le fait que, dans ce cas, chaque processeur possède sa propre unité de contrôle. Donc les processeurs sont en fonctionnement indépendant.

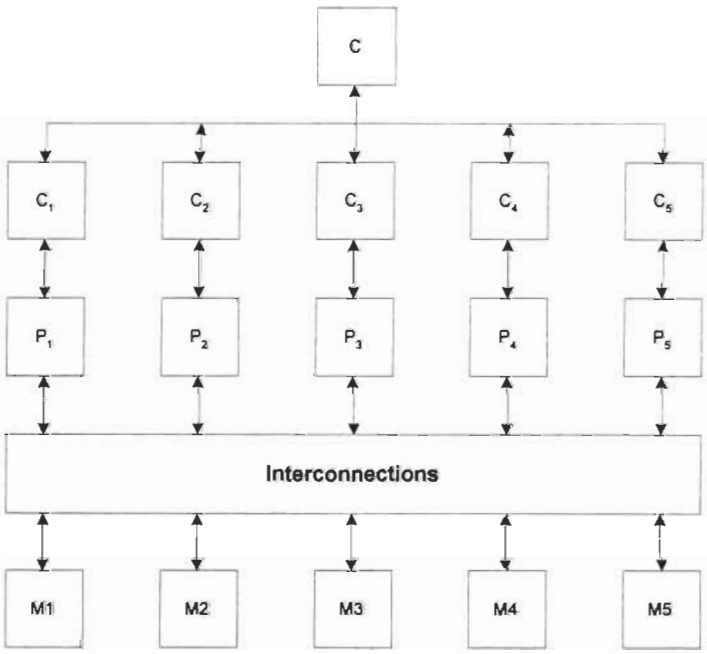


Figure 4. 3 : Architecture MIMD [COS93]

L'architecture systolique présente le meilleur exemple des systèmes massivement parallèles. Elle est constituée de cellules interconnectées localement (Figure 4.4) pouvant travailler simultanément et formant ainsi un processeur réalisant une fonction bien déterminée.

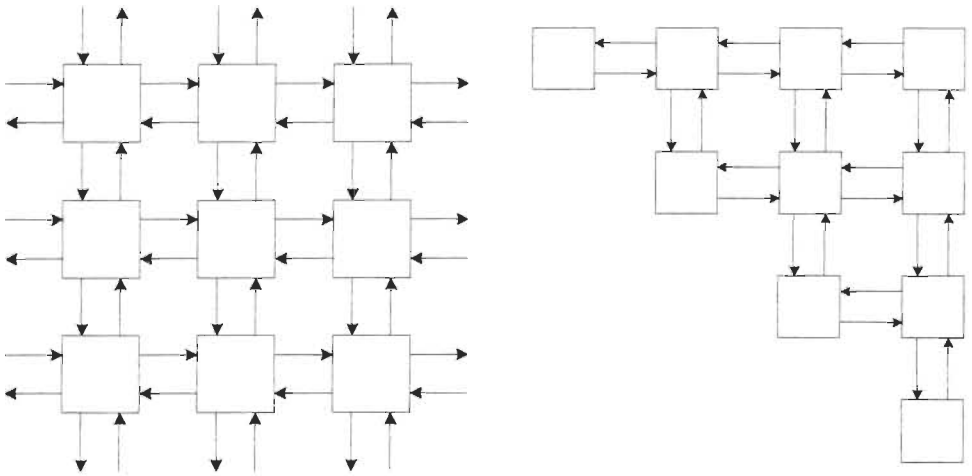


Figure 4.4 : Architectures Systoliques à topologie a) carré b) triangulaire

Dans ce chapitre nous parlerons essentiellement de la partie architecturale de l'algorithme d'égalisation à base de logique floue et sa simulation. La première section sera consacrée aux architectures systoliques et leurs caractéristiques. La seconde section portera sur le problème de quantification. Par la suite on proposera une architecture systolique pour le système d'égalisation dans le troisième paragraphe. L'avant dernier paragraphe sera consacré à l'étude d'implantation sur FPGA ou ASIC. Enfin on présentera les résultats de simulation et de synthèse.

4.1 Architecture Systolique

Les architectures systoliques sont des structures composées de processeurs élémentaires (PE) interconnectés localement. Le travail de ces cellules est synchronisé par une horloge. L'ensemble forme un réseau systolique synchrone réalisant une fonction bien déterminée. A chaque cycle d'horloge chaque PE réalise la fonction qui lui a été assigné et propage d'une manière pulsée l'information aux processeurs voisins.

Le fonctionnement du réseau systolique ressemble au phénomène de la circulation sanguine dans le corps Humain qui est pulsé par le rythme cardiaque. C'est d'ici que vient le terme systolique.

4.1.1 Propriétés des architectures systoliques

Les architectures systoliques présentent plusieurs caractéristiques qui rendent trés intéressant leur intégration en ITGE on cite :

- Simplicité et régularité
- Localité des communications
- Parallélisme massif

Cependant ce type d'architecture est souvent spécifique à une application dédiée. Donc la réalisation d'un processeur spécialisé à base de réseau systolique nécessite une unité de commande ou un processeur hôte permettant la supervision des PE. La figure 4.5 montre l'organisation d'un processeur spécialisé à base de réseau systolique.

➢ *Simplicité et régularité*

L'interconnexion locale et régulière des cellules facilite grandement l'implantation topologique. La modélisation d'un seul processeur élémentaire est suffisant pour générer tout le réseau. En plus l'optimisation d'un processeur élémentaire entraîne l'optimisation de tout le réseau, d'où un gain sur la surface d'intégration.

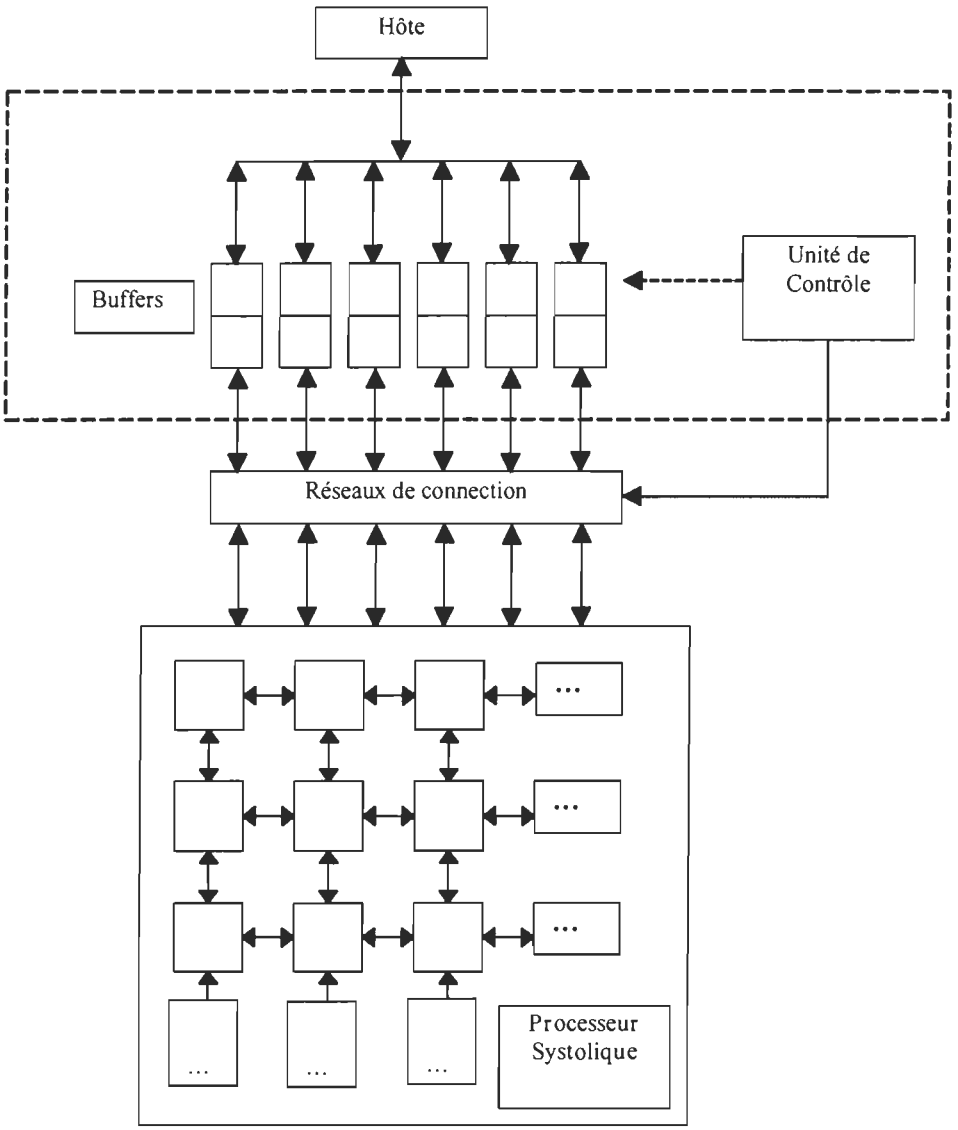


Figure 4.5 : Intégration d'un réseau systolique dans un Système [QUI89]

➢ *Localité des communications*

Généralement la présence de connections trés longue entraîne la présence de résistances et capacités parasites qui font ralentir la fréquence de fonctionnement du circuit. Dans le cas d'architecture systolique ce problème ne se pose pas, car chaque PE communique avec ses voisins immédiats amenant à des chemins de connections de longueur minimale.

➢ *Parallélisme massif*

Dans les architectures systoliques on a un réseau de PE fonctionnant simultanément et synchronisé sur la même horloge. Ce qui rend cette structure hautement parallèle. Cette structure permet d'exploiter au maximum les PE d'où une grande exploitation matérielle du processeur.

4.1.2 Différentes topologies de réseaux systoliques

Il existe plusieurs topologie d'architectures systoliques. Suivant l'algorithme on choisit la topologie qui maximise l'efficacité du réseau. Le choix de la topologie à adopter pour un algorithme donné est une tâche trés délicate qui nécessite beaucoup d'imagination et de travail. Les différentes topologies qu'on peut rencontrer dans la littérature [QUI89],[COS93] sont : les réseaux linéaires, orthogonales, triangulaires... (Figure 4.6)

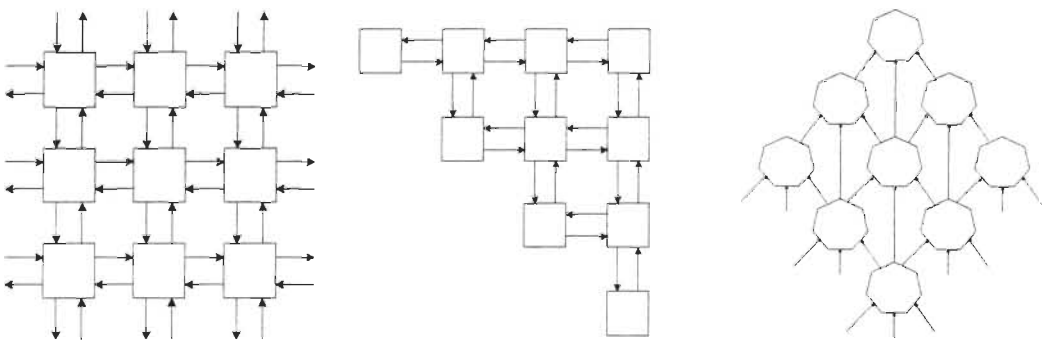


Figure 4.6 : Réseau systolique carré, triangulaire et hexagonal

4.2 Étude de quantification

De nos jours la tendance est de plus en plus vers le remplacement de l'intégration Analogique par l'intégration Numérique. L'amplitude d'un signal numérique ne peut prendre qu'un nombre fini de valeurs représentables. Si le signal à traiter n'est pas déjà numérique, il faudra le quantifier par un convertisseur analogique – numérique.

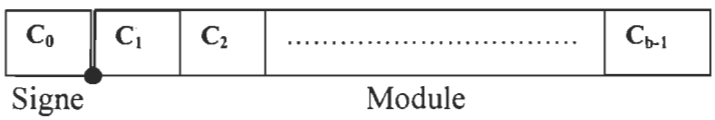
4.2.1 Méthodes de quantification

➢ *Quantification en virgule fixe*

Pour la quantification en virgule fixe le nombre de point et la distance entre les points est arbitraire, mais le plus souvent on choisit un ensemble symétrique par rapport à l'origine. Évidemment, le nombre de valeurs représentable détermine le nombre de bits qui sont nécessaires pour leur représentation. Si on a à représenter un ensemble de valeurs entre –1 et 1 sur b bits on aura les valeurs représentables suivantes :

$\{-1,\ldots, -2.2^{-(b-1)}, -2^{-(b-1)}, 0, 2^{-(b-1)}, 2.2^{-(b-1)},\ldots, 1\}$

La façon la plus naturelle est de réserver un bit de signe et le reste pour représenter le module comme fraction binaire. C'est la représentation par signe et par module.



D'où l'équation 4.1

$$valeur = (-1)^{c0} \sum_{i=1}^{b-1} c_i 2^{-i} \qquad (4.1)$$

En utilisant cette représentation on n'arrive à représenter ni –1 ni 1, et la valeur « 0 » possède deux représentations : 1 00…0 et 0 00…0. Le pas de quantification est donné par l'équation (4.2).

$$q = \frac{X_{max} - X_{min}}{2^b - 2} \tag{4.2}$$

➢ *Représentation par complément à deux*

C'est une autre représentation en virgule fixe. Elle est trés répandue, parce qu'elle présente des avantages pour les opérations arithmétiques. Elle consiste à remplacer le nombre négatif -r par 2-r et à développer ce dernier comme fraction binaire.

$$valeur \quad = c_0 + \sum_{i=1}^{b-1} c_i 2^{-i} \tag{4.3}$$

Le pas de quantification est donné par l'équation (4.4)

$$q = \frac{X_{max} - X_{min}}{2^b - 1} \tag{4.4}$$

Avec ce type de représentation on peut couvrir l'intervalle $[-1,1-2^{-(b-1)}]$ et la valeur zéro possède une seule représentation.

➢ *Représentation en virgule flottante*

Dans la représentation en virgule flottante, on consacre e bits à un facteur d'échelle sous forme d'une puissance de 2 et m=b-e-1 bits à la mantisse. Le bit restant sert soit comme bit de signe, soit comme chiffre avant la virgule si les nombres négatifs sont exprimés sous forme de complément à deux.

4.2.2 Choix de la représentation à utiliser

D'après une recherche bibliographique sur l'implantation des différents type de filtre numérique en technologie VLSI, on remarque que la représentation en virgule fixe avec en complément à 2 est la plus utilisée car d'une part, elle permet de présenter un nombre mieux qu'une représentation en virgule fixe simple et d'autre part, elle est beaucoup plus facile à implanter que la représentation en virgule flottante.

4.2.3 Choix du nombre de bit

Pour utiliser les valeurs analogiques d'entrée de l'égaliseur il faut utiliser un convertisseur analogique numérique qui permet de représenter ces valeurs sous forme binaire. Pour le choix du nombre de bits du convertisseur A/N on va procéder à la simulation suivante : on fait les simulations pour des convertisseurs à 8 et 10 bits et à chaque fois on fait varier le nombre de bits sur lequel on représente les signaux internes de 8 à 20.

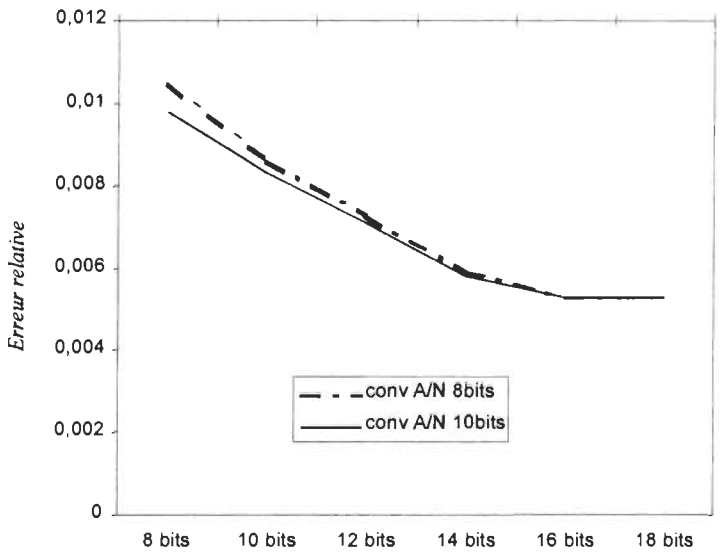


Figure 4.7a: Choix du nombre de bit du convertisseur A/N

D'après la figure 4.7a on constate que pour une représentation 16 bits des variables internes le convertisseur 8 bits et 10 bits donnent le même résultat. l'utilisation d'un convertisseur 8 bits pour la numérisation des variables d'entrée est donc suffisant.

Afin de fixer le nombre de bits sur lequel on va représenter les différentes variables internes de l'architecture on réalise la simulation de l'égaliseur dans l'environnement Matlab® en utilisant les valeurs quantifiées sur un nombre de bits variant de 8 à 24 avec un niveau de bruit allant de 7dB à 22dB. Les courbes de la figure 4.7b représentent le résultat d'une moyenne de 10 itérations chacune avec 1000 échantillons pour chaque SNR et ceci pour chaque représentation.

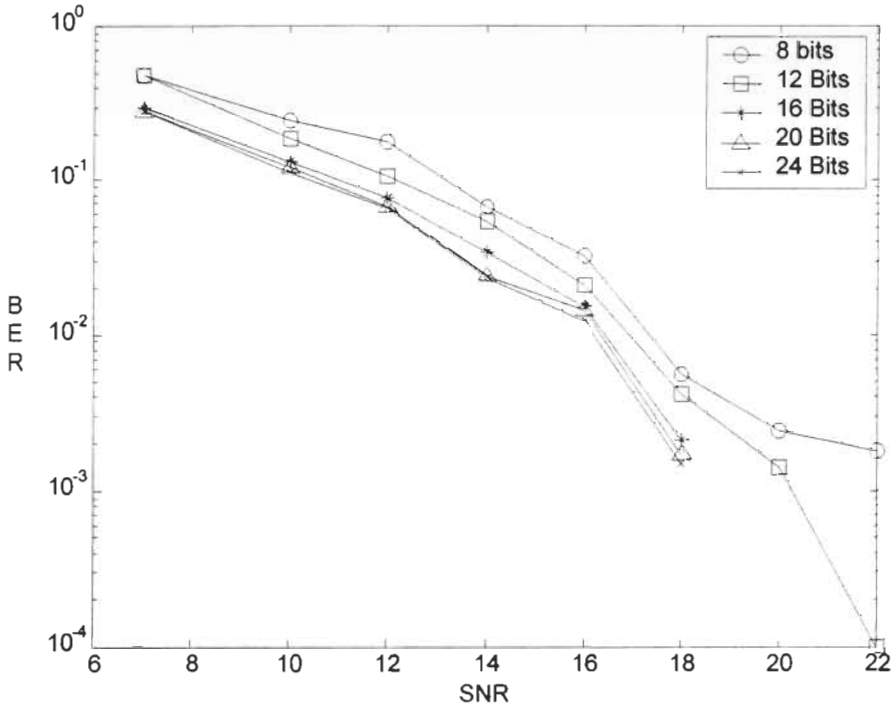


Figure 4.7b : Résultat de quantification (LF_LMS)

D'après cette simulation on remarque qu'à partir d'une représentation sur 16 bits on n'a plus d'amélioration du BER. En effet, les courbes de 16,20 et 24 bits sont presque confondues.

On présume donc qu'une représentation sur 16 bits des variables internes de l'architecture est suffisante.

4.3 Proposition d'une architecture systolique

Comme première étape de l'implantation en technologie ITGE on va considérer le cas d'un égaliseur stationnaire. Donc les paramètres du filtre à base de logique floue sont invariants et calculés à l'avance. Par suite le système d'égalisation se résumera aux équations suivantes

$$\hat{x}_k = \frac{num_k(\tilde{y})}{c_k(\tilde{\mathbf{y}})} \tag{4.5}$$

$$c_k(\tilde{\mathbf{y}}) = \sum_{l_1=1}^{m_1}\sum_{l_2=1}^{m_2}\mu(\tilde{y}_k, l_1)\mu(\tilde{y}_{k-1}, l_2) \tag{4.6}$$

$$num_k = \sum_{l_1=1}^{m_1}\sum_{l_2=1}^{m_2}\theta^{(l_1,l_2)}\mu(\tilde{y}_k, l_1)\mu(\tilde{y}_{k-1}, l_2) \tag{4.7}$$

Où k est l'indice de l'échantillon.

$$\tilde{y} = (\tilde{y}_k, \tilde{y}_{k-1})\ ;\ \theta^{(l_1,l_2)}; l_1, l_2 = 1....m_1, m_2$$

4.3.1 Architecture Systolique

L'architecture présentée sur la figure 4.8 est modélisée pour m_j =3; j=1,2. Elle permet de réaliser la fonction décrite par l'équation (4.5).

En général pour une architecture avec m_j fonctions d'appartenance (j=1,2) on estime le nombre de cellules comme suit : $m_1 \times m_2$ Processeurs élémentaires (PE), m_1+m_2 additionneurs et un diviseur.

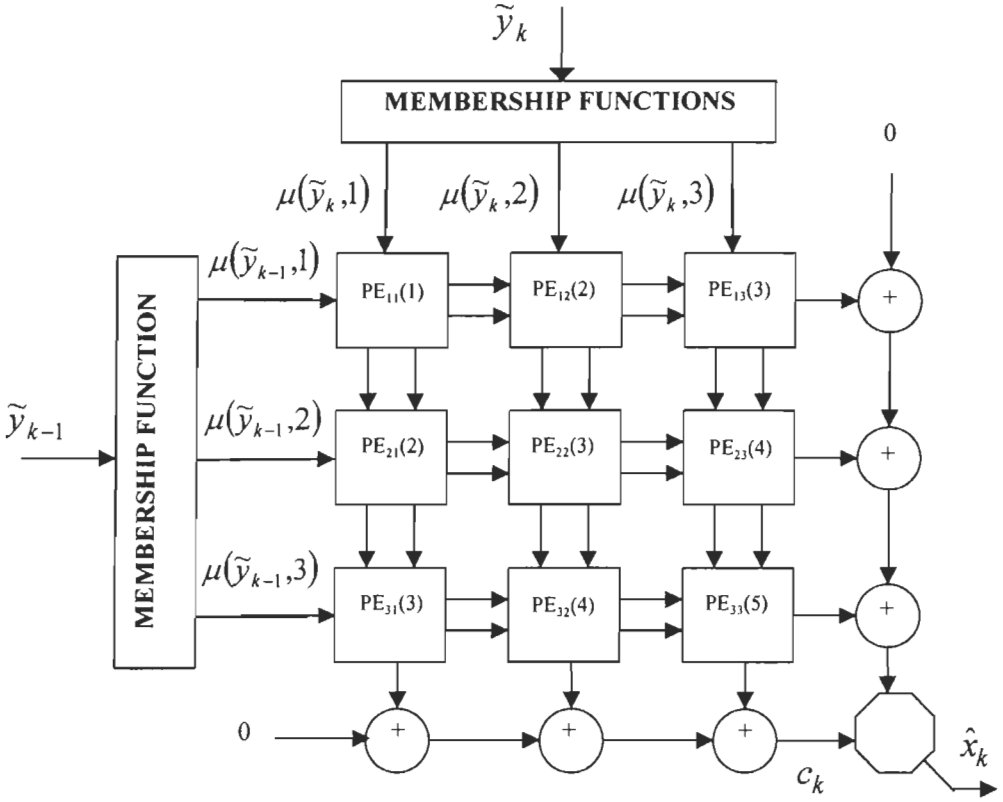


Figure 4.8 : Architecture systolique pour l'égaliseur à base de logique floue pour m_j=3 (j=1,2)

4.3.2 Description des unités opératrices

L'architecture présentée sur la figure 4.8 est composée de trois types de cellules (figure 4.9). Chacune réalise une fonction spécifique.

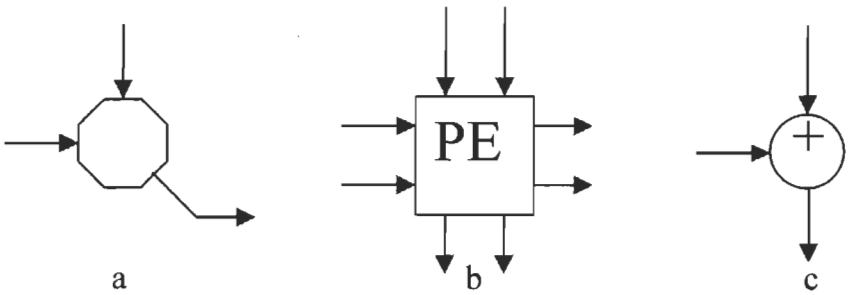


Figure 4.9 : Différentes Cellules constituants le réseau systolique

Les cellules circulaires (c) possèdent deux entrées et une sortie ils permettent de réaliser la fonction d'addition simple.

La cellule en octogonale (a) permet de réaliser la fonction de division du numérateur num_k par le dénominateur c_k.

Les cellules (b) représentent les processeurs élémentaires du réseau systoliques. On propose deux architectures internes (PE2 et PE1) pour ce type de cellules. La structure PE1 est optimisée pour la vitesse alors que PE 2 est optimisée pour la surface. L'architecture interne des processeurs élémentaires est représentée sur la figure 4.10 et 4.11. PE1 est composée de deux multiplieurs et un additionneur. PE2 a une structure plutôt complexe, cette dernière permet de réaliser la même tache que PE1 mais en trois fois le nombre de cycle. Il est composé d'un multiplieur à base d'un réseau de CSA[3] et un FA[4] (Figure 4.11). Dans cette architecture on exploite le FA pour réaliser le multiplieur en même temps il sert pour faire l'addition simple.

Au premier cycle d'horloge PE2 réalise la multiplication de $\mu(\tilde{y}_k, l_1, t-1)$ par $\mu(\tilde{y}_{k-1}, l_2, t-1)$ et met le résultat dans un registre. Au deuxième cycle d'horloge PE2 réalise la multiplication de $\theta^{(l_1,l_2)}$ par le résultat de la première multiplication. Enfin au troisième cycle on réalise l'addition du résultat obtenu par $N^{(l_1-1,l_2)}(t-1)$ et nous permet ainsi d'avoir le résultat $N^{(l_1,l_2)}(t)$. En plus PE2 transmet $\mu(\tilde{y}_k, l_1, t-1)$ et $\mu(\tilde{y}_{k-1}, l_2, t-1)$ aux processeurs voisins.

[3] CSA : c'est un additionneur à conservation de retenu en anglais « Carry Save Adder »
[4] FA : additionneur complet en anglais « Full Adder »

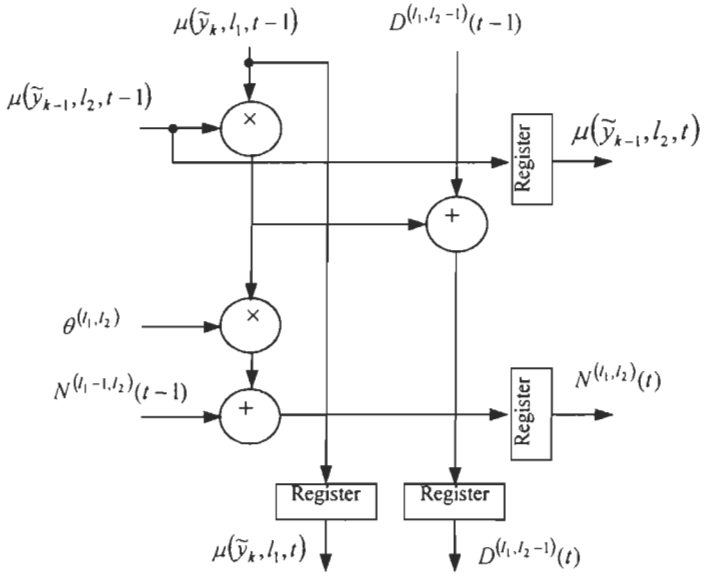


Figure 4.10 : Architecture interne de PE1

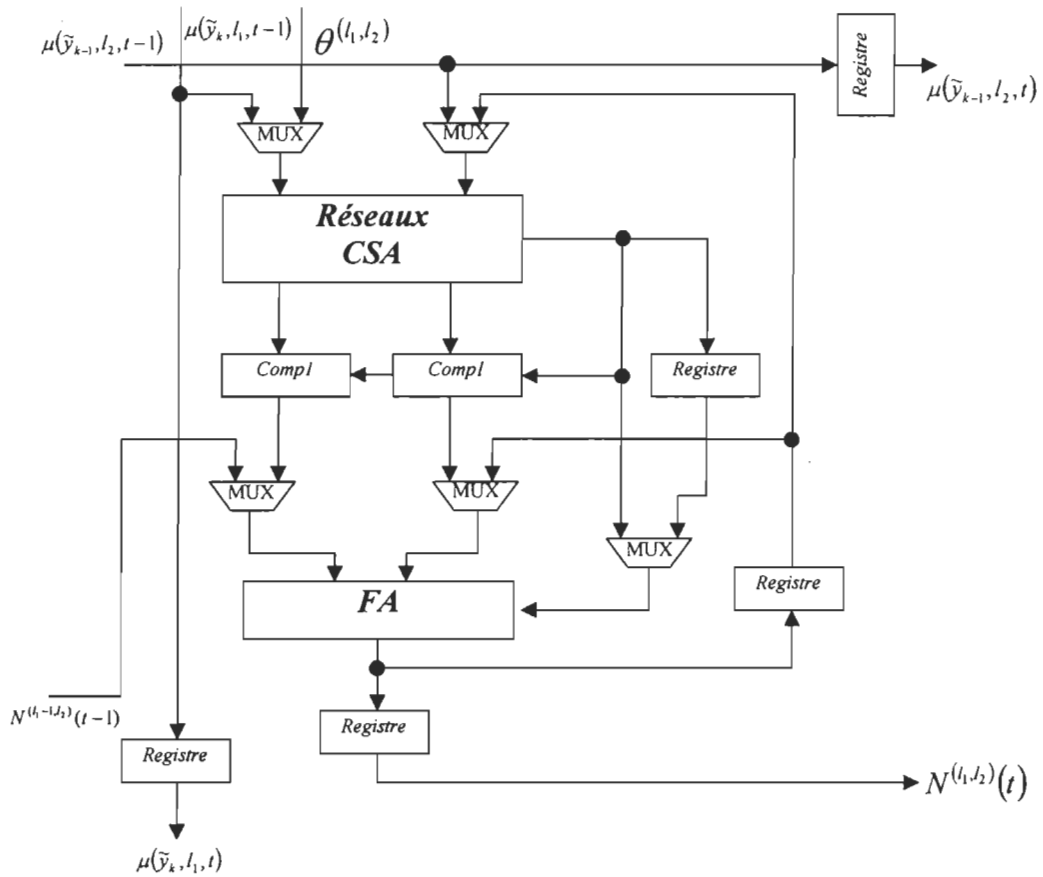


Figure 4.11 : Architecture de PE2

4.3.3 Mode de fonctionnement du réseau systolique

À chaque cycle d'horloge t le processeur $PE_{l_1,l_2}(t)$ présenté dans la figure 4.8 à pour tâche de calculer la somme partielle du numérateur $num_k(\tilde{y})$ et du dénominateur $c_k(\tilde{y})$ présenté par les équations(4.6) et (4.7) respectivement, comme suit :

$$N^{(l_1,l_2)}(t) = \theta^{(l_1,l_2)}\mu(\tilde{y}_k, l_1, \text{t-1})\mu(\tilde{y}_{k-1}, l_2, \text{t-1}) + N^{(l_1-1,l_2)}(t-1) \quad (4.8)$$

$$D^{(l_1,l_2)}(t) = \theta^{(l_1,l_2)}\mu(\tilde{y}_k, l_1, \text{t-1})\mu(\tilde{y}_{k-1}, l_2, \text{t-1}) + D^{(l_1,l_2-1)}(t-1) \quad (4.9)$$

$N^{(l_1-1,l_2)}$ et $D^{(l_1,l_2-1)}$ représentent les valeurs des sommes partielles du numérateur et du dénominateur provenant des processeurs voisins PE_{l_1,l_2-1} et PE_{l_1-1,l_2} respectivement.

En plus il dépose les valeurs de μ dans les registres correspondants. Le contenu de tous les registres est transmis au processeurs voisins au prochain cycle d'horloge. À la fin de chaque colonne ou ligne de processeurs il y a un additionneur qui a pour rôle de faire la somme du dénominateur ou numérateur. Les cellules d'addition qui se situent juste avant le diviseur contiennent la valeur du numérateur et du dénominateur. La dernière étape est de faire la division entre ces deux termes. Ce qui va donner la valeur estimée de l'échantillon d'entré.

4.3.4 Simplification de l'architecture

L'architecture ainsi présentée est limitée en fréquence par la fréquence de fonctionnement du diviseur qui constitue la dernière étape de calcul. En se référant à l'équation (4.6). On remarque que le la valeur du dénominateur est toujours positive. Donc, cette opération n'affecte pas le signe de la valeur calculée par le numérateur; par conséquent la division ne

change pas le signe du résultat final. Ainsi, le rôle de la division est de ramener les valeurs obtenues proche du couple de valeur {-1,1}. Pour simplifier l'implantation on va remplacer la cellule de division par une fonction de saturation à +1 et –1 (figure 4.12). Le modèle du système ainsi formulé est donné par la Figure 4.13.

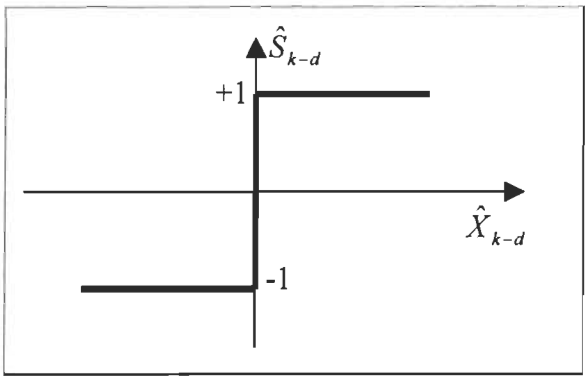


Figure 4.12 : Fonction de saturation

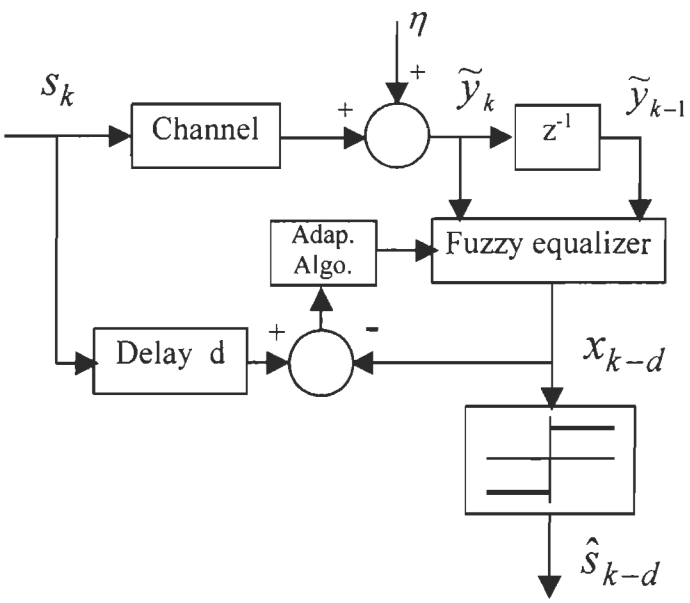


Figure 4.13 : Schéma bloque du système d'égalisation utilisant la fonction de saturation

Une fois cette simplification faite on n'aura plus à calculer la valeur du dénominateur. La figure 4.14 montre l'architecture résultante.

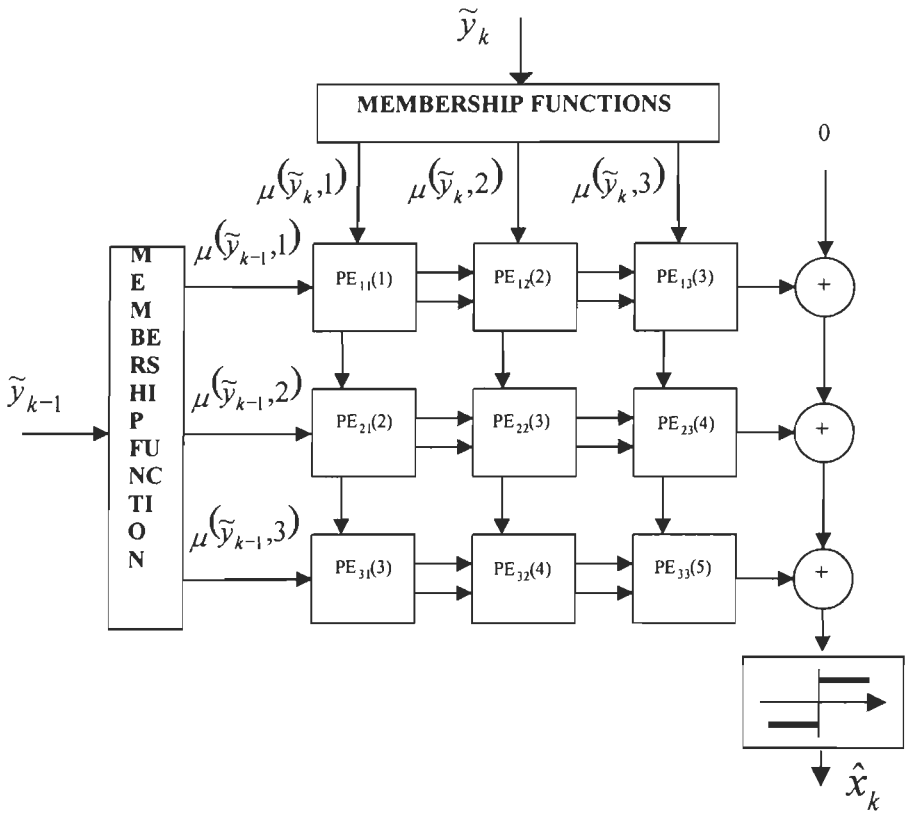


Figure 4.14 : Architecture simplifiée

(utilisation de la fonction de saturation et omission du calcul du dénominateur)

La dimension de l'architecture ainsi formulée est directement proportionnelle au carré du nombre de fonctions d'appartenance au bloc de fuzzification. Pour m_j=5, on aura un réseau systolique avec 25 processeurs élémentaires, ce qui constitue un obstacle quant-à une implantation ITGE, surtout si on est limité par la surface d'intégration. Ce facteur est trés important lorsque l'on a à faire une implantation dans un FPGA, car le nombre de portes logique est limité.

En se référant au paragraphe 3.4.3 du chapitre précédent, on remarque qu'on peut introduire d'autres simplifications sur l'architecture déjà proposée. La sortie du bloque de fuzzification est $\mu(x_k, l_1)$ et $\mu(x_{k-1}, l_2)$ avec $l_j = 1,2...m_j \quad j = 1,2$.

Pour dc=1[5] la sortie du bloc de fuzzification contiendra au maximum deux valeurs non nulles pour y_k et de même pour y_{k-1}. De ce fait on se retrouvera avec des PE qui ne font que multiplier des zéros. À chaque cycle d'horloge il y en aura quatre parmi $m_1 \times m_2$ PE, qui réaliseront un calcul utile.

En conclusion on peut remplacer le réseau systolique présenté sur la figure 4.8 par un réseau à quatre PE, en ajoutant un multiplexeur entre le bloque de fuzzification et le réseau systolique à 4 PE pour chacune des entrées y_k et y_{k-1}. (voir figure 4.15). Le rôle de chaque multiplexeur est de détecter les valeurs non nulles de la sortie du bloc de fuzzification et les transmettre au réseau systolique pour chaque cycle d'horloge.

À chaque cycle d'horloge on trouve à l'entrée de chacun des deux multiplexeurs les valeurs de $\mu(x_k, l_1)$ et $\mu(x_{k-1}, l_2)$ avec $l_j = 1,2...m_j \quad j = 1,2$. Le multiplexeur va trier parmi ces valeurs les deux valeurs non nulles qui sont toujours successives, en plus il doit fournir l'adresse des valeurs de θ allant servir au calcul pour les quatre PE.

[5] dc : degré de chevauchement des fonctions d'appartenance.

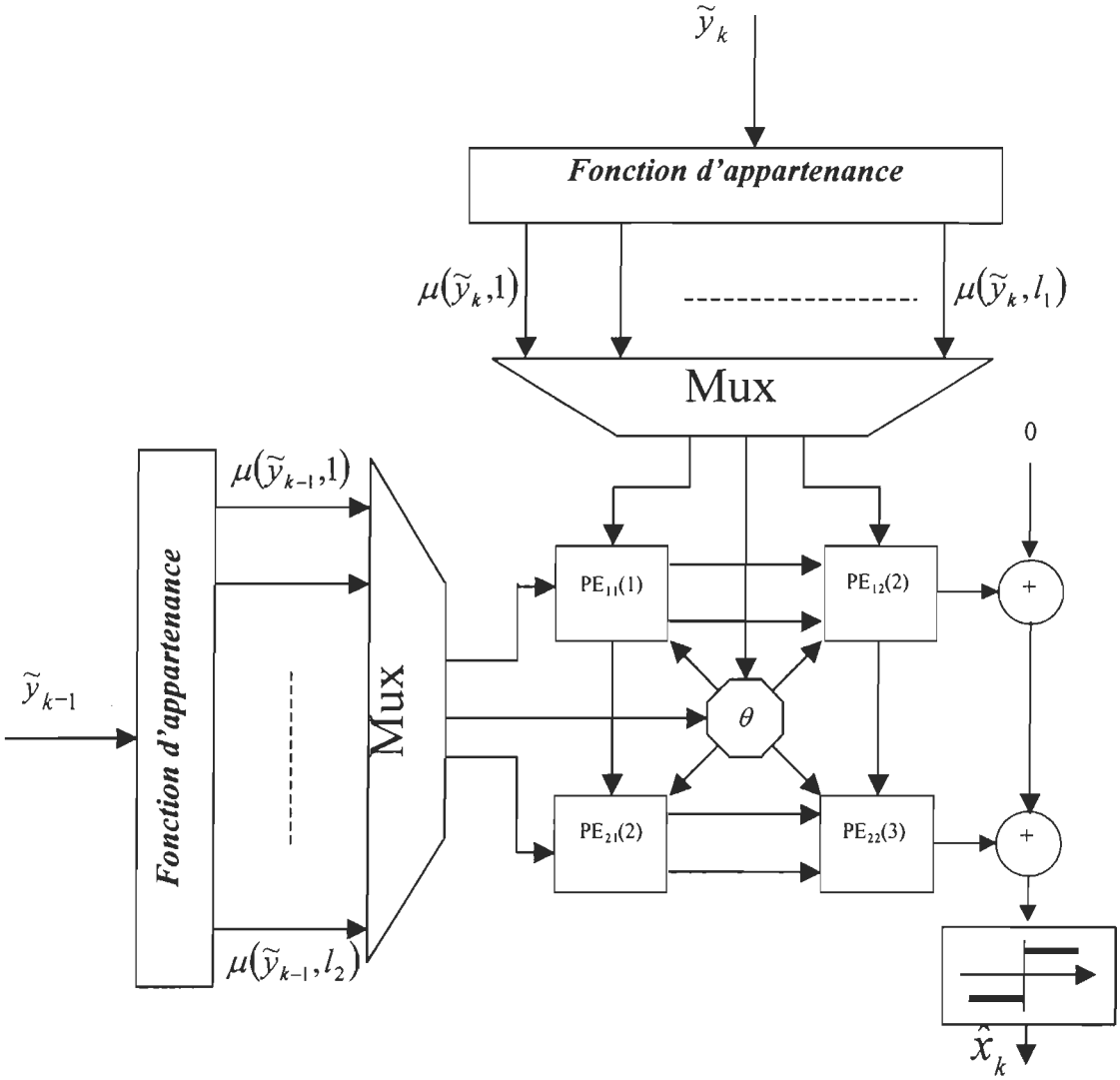


Figure 4.15 : Architecture compacte tenant compte de toutes les Simplifications

4.3.5 Proposition de l'architecture adaptative

La figure 4.16 montre la structure de l'architecture proposée. Elle est composée de 4 PE, 2 Mux, 2 additionneurs, un diviseur et un processeur central qui permet de commander les différents processeurs et séquences de fonctionnement. L'architecture ainsi présentée nous permet de faire deux tâches : égalisation de canaux et adaptation des paramètres de l'égaliseur par l'algorithme LT_LMS. Les processeurs internes ont la même structure que

PE1 sauf qu'ils vont servir maintenant pour l'adaptation aussi. Si on regarde les équations du filtre LT_LMS (Équation 3.10) on remarque que pour calculer θ_{k+1} on a besoin de faire deux multiplications et une addition. De ce fait nous pouvons exploiter la structure de nos PE pour faire ce calcul en ajoutant des multiplexeurs dans la structure interne de chaque PE, voir figure 4.17. Le rôle de ces multiplexeurs, est de fournir aux PE : soit des données pour calculer $\hat{x}_k$, c'est le mode reconstitution, soit les données pour faire la correction de θ, c'est le mode adaptation des paramètres. L'architecture de l'égaliseur de canaux adaptatif résultante est donnée par la figure 4.16. La seule différence avec les autres architectures présentées est le processeur central qui permet de synchroniser le fonctionnement des processeurs élémentaires, et de leur assigner le mode de fonctionnement approprié soit pour le calcul adaptatif de θ soit pour le calcul de le sortie de l'égaliseur $\hat{x}_k$. Lorsque le réseau systolique est en mode de reconstitution, les PE fonctionnent de la même façon que ceux décrits au paragraphe 4.3.3. Dans le mode d'adaptation des paramètres de l'égaliseur, chaque PE va recevoir les données appropriées pour faire les corrections nécessaires sur $\theta_k(i,j)$ pour avoir à la sortie de chaque PE $\theta_{k+1}(i,j)$, $(i,j=1,2,...,m_1,m_2)$. Le rôle des multiplexeurs dans l'architecture (figure 4.16) est de fournir les valeurs de $\mu(y_k,i)$ et $\mu(y_{k-1},j)$ (i=1..l_1; j=1..l_2) aux processeurs élémentaires. En fonction des valeurs du couple (i,j), les valeurs de θ servants au calcul de $\hat{x}_k$ seront choisies.

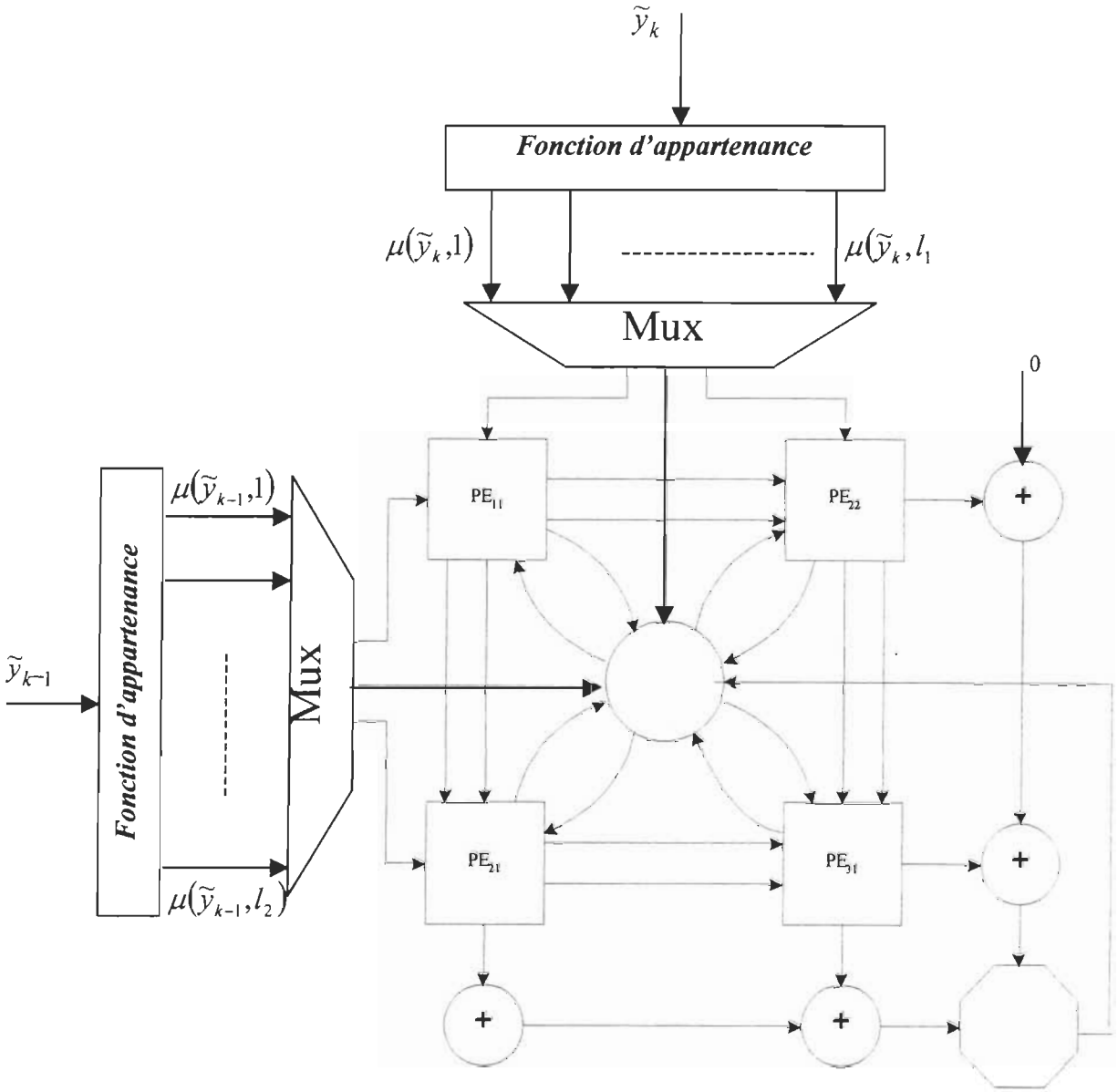


Figure 4.16 : Architecture de l'égaliseur de canaux Adaptatif

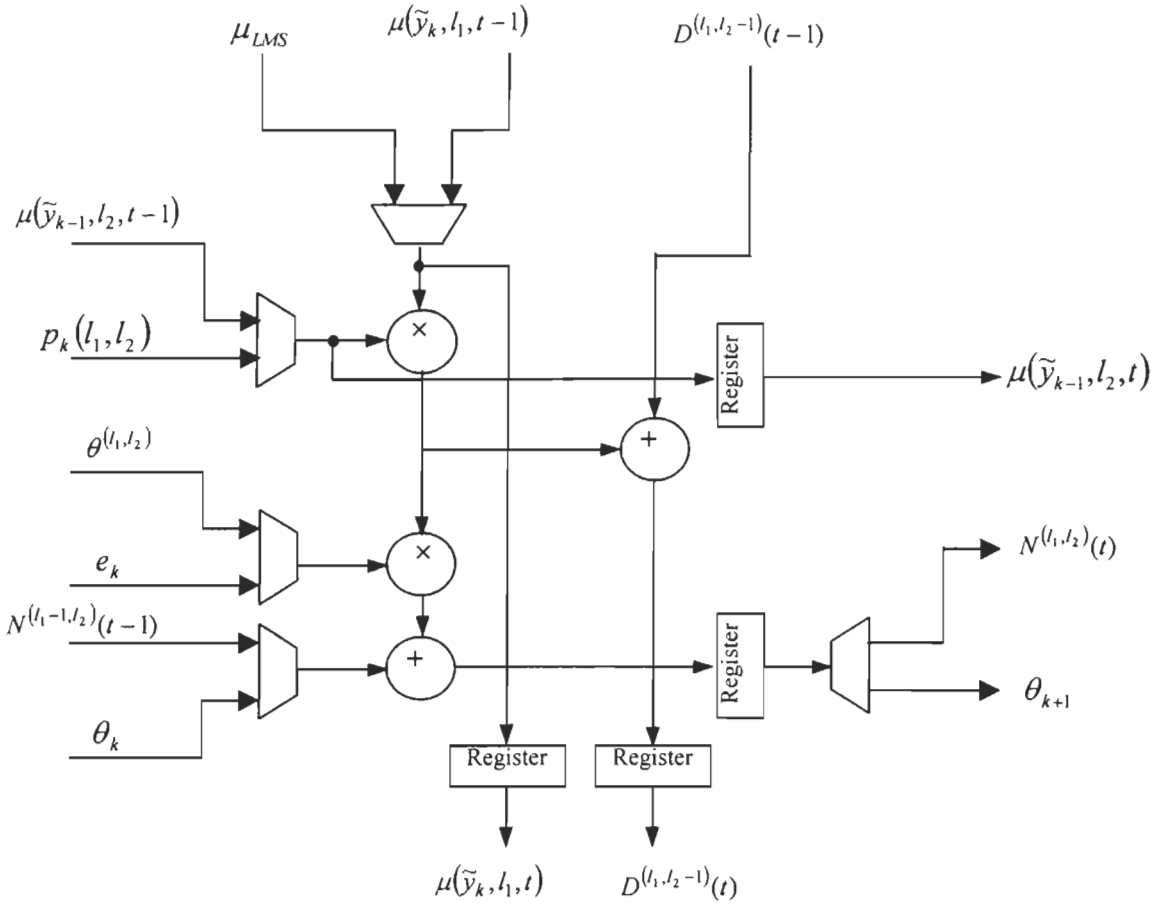


Figure 4.17 : Architecture interne de PE3

4.4 Implantation de l'architecture

4.4.1 Étapes à suivre

La conception d'un circuit intégré commence par l'adaptation de l'algorithme pour le rendre implantable, chose faite dans le chapitre 3. Ensuite le choix de l'architecture à utiliser et la définition du mode de fonctionnement des différentes cellules. Ce qui a été déjà fait dans la section 4.3. La prochaine étape est celle de la modélisation VHDL (**V**ery large scale integration circuit **H**ardware **D**escription **L**anguage) de l'architecture et sa

compilation avec le logiciel de Mentor Graphics. Ensuite on doit vérifier la fonctionnalité de l'architecture ainsi modélisée, c'est ce qu'on appelle la simulation fonctionnelle. Cette dernière tâche peut être réalisée avec le logiciel de simulation de Mentor Graphics : Qhsim. Une fois que la fonctionnalité du circuit est vérifiée on passe à l'étape de synthèse, cette dernière permet de transformer le code VHDL en porte logique correspondant à une technologie donnée (CMOS ou FPGA). Cette tâche est réalisée avec SYNOPSYS®. Enfin pour l'implantation de l'architecture dans un ASIC on utilise CADENCE® pour faire le layout. Pour le cas des FPGA on utilise les outils de placement et routage fournis par le constructeur.

4.4.2 Modélisation VHDL

C'est un langage complet de description matérielle, il a trouvé ses premières applications dans la modélisation et la simulation. Il permet, à partir d'une description textuelle de décrire le fonctionnement d'un circuit donné.

Le réseau systolique de la figure 4.15 a été modélisé en VHDL. Certaines cellules de l'architecture on été faites en structurelle, d'autres en comportementale. (voir annexes B)

4.4.3 Simulation fonctionnelle

La simulation fonctionnelle est celle qui permet de valider le fonctionnement de l'architecture modélisée en VHDL. Pour vérifier le fonctionnement du circuit on utilise généralement un fichier dans lequel on précise les valeurs des entrées pour récupérer la sortie et la comparer avec les valeurs données par Matlab. Ce fichier est appelé «forcefile», la génération du force file est faite par un programme Matlab (voir Annexe C).

Le modèle VHDL de l'architecture, présenté sur la figure 4.8 à été simulé par Qhsim de Mentor Graphics avec l'utilisation des données synthétiques avec le canal linéaire défini par l'équation (4.10).

$$\tilde{y}_k = s_k + 0.5 s_{k-1} + \eta_k \qquad (4.10)$$

Les résultats de simulation sont présentés à la figure 4.18

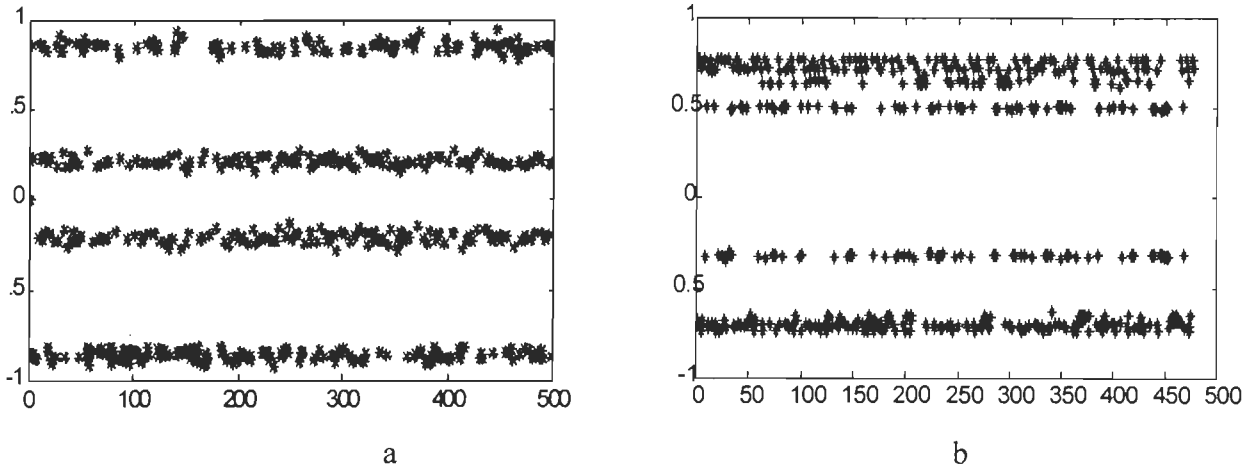


Figure 4.18 : Résultat de simulation du code VHDL

a. Signal de sortie du canal b. Signal à la sortie de l'égaliseur

On remarque que la simulation du modèle VHDL a fourni les résultas attendus. L'algorithme a réussi à séparer les 1 et les −1. On a un BER=50% sur la figure 4.18-a, après égalisation on obtient un signal avec un BER=0% (figure 4.18-b). Plus de détails sont dans l'article de conférence [ZAK99b]. Ceci valide le modèle VHDL utilisé.

4.4.4 Synthèse en technologie CMOS

la synthèse sur Synopsys de l'architecture présentée sur la figure 4.15 utilisant les deux types de processeurs élémentaires : PE1 et PE2 ont été fait en technologie CMOS 0.5 µm. les résultas de synthèse sont résumés par le tableau 4.1.

Tableau 4.1 : Résultat de synthèse

	SURFACE (EN NOMBRE DE TRANSISTOR)	*DÉBIT (CYCLES)*	*FRÉQUENCE DE FONCTIONNEMENT(MHZ)*	*DÉBIT (MHZ)*
Architecture systolique utilisant PE1	*20 730*	*1*	*41 MHz*	*41 MHz*
Architecture systolique utilisant PE2	*12 394*	*3*	*88 MHz*	*29 MHz*
Architecture avec 9 PE1 (figure 4.8)	*80 000*	*1*	*40 MHz*	*40 MHz*

4.4.5 Synthèse en technologie FPGA

Le FPGA (FPGA : Field Programmable Gate Array), est une version plus évoluée et offrant plus de flexibilité que les PLD. Dans le cadre de ce mémoire on va utiliser le FPGA de la compagnie Xilinx (XC4036EX) qui a une capacité équivalente d'environ 35000 portes logiques. Le FPGA est souvent utilisé pour créer un circuit intégré prototype dans la phase préliminaire du développement d'un ASIC, et ceci en raison de sa facilité de programmation ce qui nous fait gagner le temps de développement.

Les FPGA de Xilinx sont composés de cellules programmables les IOB et les CLB.

Les IOB (Input Output Blocs) qui sont comme tampons aux entrées/sorties de notre FPGA.

Et les CLB (Control Logic Blocs) qui sont des cellules programmables pour réaliser les

fonctions combinatoires et séquentielles de notre architecture à implanter. Le FPGA dont on dispose contient 288 IOB et 1296 CLB.

Pour l'implantation FPGA on a commencé par le même point de départ que l'implantation en technologie CMOS c'est-à-dire le programme VHDL. Ensuite on a utilisé Synopsis pour la synthèse en ciblant la technologie FPGA XC4036EX d'Xilinx. Cette étape va nous permettre de convertir (compiler) la description VHDL en format XNF. Ce fichier est ensuite utilisé par Design Manager de Xilinx pour fournir un fichier en format BIT. C'est ce dernier qui va servir à la configuration du FPGA avec notre architecture. Dans notre cas on utilise la carte Aristotle de mirotech pour l'implantation hardware. Elle est composée d'un DSP TMS320c44 de Texas Instruments et du FPGA XC4036EX d'Xilinx (voir figure 4.19)

Figure 4.19 : l'architecture de la carte Aristotle de Mirotech

Les résultats de synthèse après placement et routage par Design Manager de Xilinx utilisants le FPGA XC4036EX sont résumés dans le tableau 4.2.

Tableau 4.2 Résultats de Synthèse sur FPGA

	CLB utilisés (1296)	*IOB utilisés (288)*	*Analyse temporelle après layout*		
			Fréquence	*Débit*	*latence*
Architecture utilisant PE1	*535 (41%)*	*130 (45%)*	*8 MHz*	*Chaque cycle*	*5 cycles*
Architecture utilisant PE2	*309 (23%)*	*106 (36%)*	*15 MHz*	*Chaque 3 cycles*	*15 cycles*

Si on compare le résultas du tableau 4.2 avec 4.1 on remarque que la même architecture implantée sur ASIC roule 5 fois plus vite que sur un FPGA.

L'implantation physique de l'architecture sur le FPGA se fait par un programme en langage C (Annexe D), et une interface graphique permettant de communiquer avec la carte qui est installé sur un slot PCI. Cet interface nous permet, à travers le programme C, de charger le fichier (.bit) dans le FPGA.

4.5 Conclusion

Au niveau de ce chapitre nous avons pu proposer une architecture pour l'égalisation de canaux stationnaires qu'on a modélisé en VHDL. Ensuite nous avons fait la synthèse en deux technologies différentes (CMOS et FPGA). La performance de l'architecture est donnée en terme de Latence et débit. La latence des architectures stationnaires est de m_1+m_2+1 et une fois que le réseau systolique est plein on aura un résultat à chaque cycle. La latence et le débit de l'architecture adaptative vont dépendre du nombre de cycle d'adaptation des paramètres.

L'utilisation des circuits matriciels FPGA ayant la même philosophie que les PLA (Programmable Logic Array), accélère énormément le processus du développement des circuits spécialisés. Une fois familiarisé avec les outils de synthèse la conception d'un processeur spécialisé sur FPGA peut prendre quelques jours. Alors que la fabrication d'un ASIC peut prendre quelques mois. L'utilisation des FPGA est trés intéressante pour la réalisation de prototypes. Par contre les ASICs peuvent atteindre des fréquences de fonctionnement beaucoup plus importantes. En se référant aux tableaux 4.1 et 4.2 on remarque que l'architecture implantée sur ASIC fonctionne avec une fréquence d'horloge 5 fois plus rapide que celle implantée sur FPGA.

Chapitre 5

Conclusion Générale

Les systèmes de communication sont appelés à se développer de façon considérable. Outre la téléphonie classique, les liaisons hertziennes seront utilisées dans un avenir proche dans des contextes aussi variés que la diffusion, les transmissions haut-débit intra-batiments (Ethernet Gigabit hertzien), etc. Les transmissions à haut débit sur câble sont également l'objet d'un intérêt important. Les spécifications en termes de performance (bande passante) et de mobilité des nouveaux systèmes de communication posent de véritables défis technologiques. Ce qui a relancé l'intérêt des recherches concernant les techniques de transmission. Le traitement du signal est appelé à jouer un rôle central pour répondre à ces défis. La problématique qu'on soulève dans le cadre de ce mémoire est celle de l'égalisation de canaux. Cette technique nous permet une exploitation optimale des médiums de transmissions. En effet, la bande passante du canal de transmission limite la vitesse de transmission des données. L'égalisation de canaux nous permet d'aller au-delà de ces

frontières physiques. La réalisation de cette étude nous a permis d'explorer une nouvelle méthode d'égalisation de canaux, celle se basant sur la logique floue. La logique floue compte parmi les nouvelles techniques de traitement des systèmes. Elle a eu ces premières applications dans le domaine du contrôle et du commande des systèmes. Récemment, on commence à s'intéresser à utiliser cette technique dans le domaine des télécommunications. Se référant à ce travail de mémoire, on peut dire que la logique floue a donné des résultats intéressants lors de son application à l'égalisation de canaux. Une étude comparative entre les différentes méthodes d'égalisation au laboratoire LSSI[6] a démontré que l'algorithme à base de logique floue est classé parmi les trois algorithmes les plus performants parmi 15 méthodes étudiées.

Au niveau de ce mémoire, on a commencé par valider le fonctionnement de l'algorithme proposé par Wang & Mandel, qui constitue la base de toute cette étude. Ensuite on a proposé des simplifications, qu'on a validé, sur l'algorithme afin de le préparer pour une implantation en technologie ITGE. La suite des travaux était de proposer une architecture systolique pour le modèle stationnaire, auquel on a introduit plusieurs simplifications pour satisfaire le compromis surface d'intégration fréquence de fonctionnement. Après une synthèse en technologie CMOS et FPGA sur Synopsis®. La première architecture proposée pour l'égaliseur à base de logique floue nous a permis d'atteindre une fréquence d'horloge de 40Mhz avec un débit à chaque cycle. La surface d'intégration est équivalente à 80000 transistors en technologie 0.5µm en technologie CMOS. La simplification de l'architecture nous a permis d'arriver avec une architecture plus performante utilisant seulement 20000

[6] LSSI: Laboratoire de Signaux et Systèmes Intégrés à l'université du Québec à Trois-Rivières.

transistors et fonctionnant à la même fréquence d'horloge, soit 40 Mhz et un débit chaque cycle.

L'implantation FPGA de cette même architecture, nous a permis d'atteindre une fréquence de fonctionnement de 8Mhz avec un débit à chaque cycle.

On peut dire que l'étude faite dans ce mémoire constitue une étape importante et incontournable pour la préparation de l'implantation d'algorithme d'égalisation de canaux à base de logique floue. C'est le point de départ de toute autre étude comparable.

Ce projet nous a permis de participer à une conférence à caractère internationale [ZAK99a] et un symposium [ZAK99b]. Le papier de conférence présenté à la conférence CCECE'99 de IEEE "IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'99)", avait pour but de présenter une première version de l'architecture systolique dédiée à l'égaliseur de canaux, basée sur la logique floue (Annexe E) . Au symposium TEXPO'99 on a proposé un poster (Annexe F) avec l'architecture simplifiée et sa mise en œuvre sur le FPGA XC4036EX de la compagnie Xilinx.

La suite de ce travail serait la modélisation VHDL de la partie adaptative associée à l'égaliseur de canaux à base de logique floue, ainsi que l'étude de l'implantation de l'égaliseur adaptatif complexe.

Bibliographie

[AMA96] Jean-Luis AMAT, Gérard Yahiaoui, Techniques avancées pour le traitement de l'information, Cépadués- éditions 1996.

[ASC96] Giuseppe Ascia, Vincenzo Catania, Marco Russo, Lorenzo Vita, "Rule-Driven VLSI Fuzzy Processor", IEEE Micro. JUNE 1996

[AUM97] Michel AUMIAUX, Initiation au langage VHDL, MASSON,1997

[BRO97] J-M. Brossier, Signal et communication numérique, égalisation et synchronisation, HERMES,1997

[CAT94] Vincenzo Catania, Antonio Puliafito, Marco Russo and Lorenzo Vita, "A VLSI Fuzzy Inference Processor Based on a Discrete Analog Approach", IEEE Transaction on fuzzy systems. Vol. 2, No. 2, MAY 1994

[CHO93] Choi, J.; Bang, S.H.; Sheu, B.J, "A programmable analog VLSI neural network processor for communication receivers". IEEE Transactions on

Neural Networks. May 1993.

[CIV95] P. L. Civera, D. Demarchi and G. Masera, "All-digital VLSI fuzzy inference engine : a case study", INT. J. ELECTRONIC Vol. 79, No. 2, 193-203, 1995.

[COH98] Ben Cohen, VHDL, Answers to frequently asked Questuion, 2eme édition Kluwer academic publishers 1997.

[COS93] Michel Cosnard, Denise Trystram, Algorithmes et architectures parallèles, 1993, interEdition, Paris

[EDD95] Rabel Claude Eddy, Réalisation d'un processeur RISC pour la logique floue, Thèse, 1995, Université du Québec à Trois-Rivières

[GLA96] Alain Glavieux / Michel Joindot, Communication numériques Introduction, Masson, Paris, 1996

[HAY96] Simon Haykin, Adaptive filter theory – 3eme édition

[JAM98] Hassan Jamali, Yves Coutu, Communication Numérique et réseaux, édition Reynald Goulet 1998.

[JER97] Ahmed Amine Jerraya, Hong Ding, Polen Kission, Maher Rahmouni, Behavioral Synthesis and Component Reuse with VHDL, KLUWER ACADEMIC PUBLISHERS,1997

[JIA96] Jian-Jun Xue , Xiao-HU Yu, "A mean field annealing partially-connected neural equalizer for pan-Eropean GSM system", p. 701 – 705 vol. 2, IEEE

international conférence on communication, 1996. ICC'96 23-27 june 1996.

[LAR97] Philippe Larcher, VHDL Introduction à la synthèse logique, édition Eyrolles 1997.

[LEE94] K. Y. Lee, "Fuzzy adaptive decision feedback equaliser, ELECTRONICS LETTERS. Vol. 30, No. 10, 12th MAY 1994

[MAD95] Vijay K. Madisetti, VLSI Digital signal processors, 1995

[MAR95] Xavier Marsault, Compression et recyclage des données multimédias - 2eme édition, Hermès, Paris, 1992, 1995

[MAS95] Daniel Massicotte, Une approche à l'implantation en technologie VLSI d'une classe d'algorithmes de reconstitution de signaux, Thèse du diplôme de Ph.D. 1995.

[MAU97] Maurice Rivoire, Jean-luis Ferrier. Commande par régulateur d'identification, edition Eyrolles 1997

[MOO96] Jason Moore, Sanders, Lockheed-Martin, Mahmoud A. Manzoul, "An Interactive Fuzzy Cad Tool", IEEE Micro. APRIL1996

[PAT98] Sarat Kumar Patra, "Efficent architecture for bayesian equalization using fuzzy filters". IEEE on circuit and systems Vol. 47, No. 7, July1998

[PEL97] David Pellerin, Douglas Taylor , VHDL Made Easy, PRENTICE HALL PTR,1997.

[PRO95] John G. Proakis, Digital communications - 3eme édition

[QUI89] P. Quinton, Y. Robert, Algoritmes et architectures systoliques, Masson, Paris, 198

[SAL95] Luis de Salvador, julio Gutiérrez, "A Multilevel Systolic Approach for Fuzzy Inference Hardware", IEEE Micro. October 1995

[SAR95] P. Sarwal and M. D Srinath, "A Fuzzy Logic System for Channel Equalization"IEEE Transaction on fuzzy systems. Vol. 3, No. 2, MAY 1995

[STO96] J. A. Stover, D. L Hall and R. E. Gibson, "A Fuzzy-Logic Architecture for Autonomous Multisensor Data Fusion", IEEE Transaction on industrial electronics. Vol. 43, No. 3, JUNE 1996

[SUR95] Hartmut Surmann, Ansgar P. Ungering, "Fuzzy Rule-Based Systems on General-Purpose Processors", IEEE Micro. August 1995

[VID99] Martin Vidal, Daniel Massicotte, "A VLSI parallele architecture of a piecewise linear neural network for non-linear channel equalization" p. 1629 – 1634 Instrumentation and measurment technologie of the 16th IEEE May 24-26,1999

[WAN92] Li-Xin Wang and Jerry M. Mendel, "Fuzzy Basis Functions, Universal Approximation, and Orthogonal Least-Squares Learning", IEEE Transaction on neural networks. Vol. 3, No. 5, September 1992

[WAN93] L.-X. Wang and J. M. Mendel, "Fuzzy Adaptative Filter With Application to Nonlinear Channel Equalization", IEEE Transactions on Fuzzy System,

to Nonlinear Channel Equalization", IEEE Transactions on Fuzzy System, Vol. 1, No. 3, August 1993, pp. 161-170.

[WEB97] Jacques Weber, Maurice Meaudre, VHDL. Du langage au circuit, du circuit au langage. MASSON, 1997

[XUE96] Jian-Jun Xue; Xiao-Hu Yu " A mean field annealing partially-connected neural equalizer for pan-European GSM system ", IEEE International Conference on Communications, ICC '96.

[ZAD65] Zadeh, Lotfi A. "Fuzzy sets." Information and Control 8 (1965): 338-353.

[ZAD75] Zadeh, Lofti A. "The concept of a linguistic variable and its application to approximate reasoning." Information Sciences 8 (1975):199-249.

[ZAK99a] Zakhama Mourad, Daniel Massicotte "FPGA implantation of systolic architecture for channel equalization using fuzzy logic", TEXPO'99 june 99

[ZAK99b] Zakhama Mourad, Daniel Massicotte "A Systolic Architecture for Channel Equalization Based on a Piecewise Linear Fuzzy Logic Algorithm", IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'99) Edmonton, Alberta, Canada, 9-12 may, 1999

ANNEXES

Annexes A

Programmes Matlab pour la simulation
de LF_RLS (annexe A1)
et LF_LMS (annexe A2)
programme Matlab pour l'étude de
Quantification (annexe A3)

```
%*************************************************************
%                  Programme principal                       *
%  Programme Matlab pour la simulation de l'égaliseur        *
%  Adaptatif à base de la logique floue                      *
%  il utilise les fonctions suivantes:                       *
%  p.m pour le calcul du vecteur p                           *
%  bruit.m génération du bruit avec une vriance donnée       *
%  tri.m pour la génération des fonctions d'appartenances    *
%  de forme CLM piecewise.                                   *
%*************************************************************
clear;
close all;
%*************************************************************
%     INITIALISATION DES VARIABLES                           *
%*************************************************************
lamda=0.91;
sigma=0.01;
ordre=150;  % nombre d'iteration necessaire pour l'adaptation du filtre
m=9;
var_bruit=0.09;

%*************************************************************
%*************** generation de signal carre ****************
%*************************************************************

s=rand(500,1);
l=length(s);
for i=1:l, if s(i)>=0.5, s(i)=1;, else s(i)=-1;, end, end

%*************************************************************
%***********generation de bruit de varience donnee *********
%*************************************************************

bruit=bruit(l,var_bruit);

%*************************************************************
%***************** effet du canal ************************
%*************************************************************

for k=2:1:l

   x(k)=s(k)+0.5*s(k-1)-0.9*(s(k)+0.5*s(k-1))^3;
   xb(k)=x(k)+bruit(k);
end;

%*************************************************************
%***** initialisation des parametres de l'equalizeur *******
%*************************************************************

fk(1)=0;
teta=[];
   for k=1:1:(m*m*m)
   teta=[teta;0.1]; %(rand-0.5)/100];
   end;
```

```
tetaK_1=teta;
P=eye(m*m*m);
PK_1=P*sigma;
figure(4);

%************************************************************
%********* adaptation des parametres de l'equaliseur *******
%************************************************************

for i=3:1:ordre
   flops(0);
   p_x=p(xb(i),xb(i-1),xb(i-2),m);

   phi=p_x;

   PK=(1/lamda)*(PK_1-(PK_1*phi*((lamda+phi'*PK_1*phi)^(-
1))*(phi'*PK_1)));

   K=PK_1*phi*(lamda+phi'*PK_1*phi)^(-1);


   tetaK=tetaK_1+K*(s(i)-phi'*tetaK_1);

   fk(i)= p_x'*tetaK;

   er_s(i)=s(i) -fk(i);

   PK_1=PK;

   tetaK_1=tetaK;
   tex(:,i)=tetaK;


   semilogy(abs(er_s)), pause(0.001);
nbre_operation_par_cycle_adaptation = flops;
end;
nbre_operation_par_cycle_adaptation

%************************************************************
%********* cycles de reconstitution sans adaptation *********
%************************************************************
mx=0;
for k= ordre+2:1:l
flops(0);
   mx=mx+1;
   p_x=p(xb(k),xb(k-1),xb(k-2),m);

   fk(k)= p_x'*tetaK;

   erreur_s(mx)=s(k) -fk(k);

nbre_operation_par_cycle_de_reconst = flops;
```

```
end;

err_absolue = norm(erreur_s)/(1-ordre)
nbre_operation_par_cycle_de_reconst

err_relatif = norm(erreur_s)/norm(s)

%************************************************************
%******************** liste des figures *********************
%************************************************************

figure(1);
plot(s,'*');
title('signal d entree du canal');
xlabel=('echantillons');
ylabel=('amplitude');
grid;
axis([0 1 -1.5 1.5]);

figure(2);
semilogy(abs(er_s));
grid;

figure(3);
plot(xb,'*');
title=('signal de sortie du canal + bruit');
xlabel=('amplitude');
ylabal=('echantillons');
grid;
axis([0 1 -2 2]);
zoom;

figure(5);
plot(fk,'*');
grid;
zoom;
grid;

k=50:1:1;
figure(6);
plot(fk(k),fk(k-1),'*');
xlabel=('x_k');
ylabel=('x_k_-_1');
grid;

figure(8)
plot(xb,'*')
grid;


figure(11);
plot(er_s);

%************************************************************
```

```
%************* calcul du rapport signal/bruit **************
%*********************************************************

s_b=20*log10(norm(s)/norm(bruit))
s_b_adap=20*log10(norm(s)/norm(bruit))
figure(12);
plot(tex','*');
zoom;

%****************************************
%           Fonction p.m                *
%****************************************
function [p_x] = p(x1,x2,x3,m);
den=0;
n=0;
num=0;
for m3=1:1:m

for m2=1:1:m

   for m1=1:1:m
      n=n+1;
      den=den+tri(x1,m1,m)*tri(x2,m2,m)*tri(x3,m3,m);
      num(n)=(tri(x1,m1,m)*tri(x2,m2,m)*tri(x3,m3,m));
   end;
end;
end;
p_x1=num/den;
p_x=p_x1';



%*********************************************
%        Fonction tri.m                      *
%*********************************************

function [triangle] = tri(x,n,m)

if m == 9
xj=[-2 -1.5 -1 -0.5 0 0.5 1 1.5 2];
elseif m==5
xj=[-2 -1 0 1 2];
end;
xm=x-xj(n);

y=0;

if   xm <= 0
   if xm >= -0.5
      y=2*xm+1;
   else
      y=0;
   end;
elseif   xm>=0
```

```
   if xm<=0.5
      y=-2*xm+1;
   else
      y=0;
   end;

end;

triangle=y;


%*********************************************
% fonction pour la génération des fonctions
% d'appartenances gaussienne
%*********************************************
function [mu_Fij] = mu(xi,n,m1,m2)

   if m1==9
   xj=[-2 -1.5 -1 -0.5 0 0.5 1 1.5 2];
   elseif m1==5
   xj=[-2 -1 0 1 2];
   end;
   xjj=xj(n);
   mu_Fij= exp(-0.5*((xi-xjj)/0.3)^2);


%*****************************************
%        Fonction bruit.m                 *
%*****************************************
function [br] =bruit(l,variance)
b=randn(l,1);
bruit=b*(variance)/std(b);
br=bruit;
```

```
%*********************************************************
% programme Matlab pour la simulation de l'égaliseur à
% base de logique floue utilisant LMS pour l'adaptation
% des paramètres.
%*********************************************************
clear;
close all;
%*********************************************************
%     INITIALISATION DES VARIABLES                       *
%*********************************************************
lamda=0.91;
sigma=0.1;
ordre=300;  % nombre d'iteration necessaire pour l'adaptation du filtre
m1=9;
m2=9;
var_bruit=0.1;
mu=0.35;
NBE=0;
%*************************************************************
%*************** generation de signal carre ******************
%*************************************************************

s=rand(1000,1);
l=length(s);
for i=1:l, if s(i)>=0.5, s(i)=1;, else s(i)=-1;, end, end

%*************************************************************
%***********generation de bruit de varience donnee *********
%*************************************************************

bruit=bruit(l,var_bruit);

%*************************************************************
%***************** effet du canal ************************
%*************************************************************

for k=2:1:l

   x(k)=s(k)+0.5*s(k-1)-0.9*(s(k)+0.5*s(k-1))^3;
   xb(k)=x(k)+bruit(k);
end;

%*************************************************************
%***** initialisation des parametres de l'equalizeur *******
%*************************************************************

fk(1)=0;
teta=[];
   for k=1:1:(m1*m2)
   teta=[teta;0.1]; %(rand-0.5)/100];
   end;

tetaK_1=teta;
P=eye(m1*m2);
```

```
PK_1=P*sigma;
figure(4);


%************************************************************
%********* adaptation des parametres de 1'equaliseur *******
%************************************************************

teta=zeros((m1*m2),1);
for i=10:1:ordre

   p_x=p(xb(i),xb(i-1),m1,m2);

   fk(i)= p_x'*teta;

   e(i)=s(i)-fk(i);

   teta1= teta+mu*e(i)*p_x;

   teta=teta1;

   if fk(i)<0 fkb(i)=-1;
   else fkb(i)=1;
   end;
   erreur_de_bit=s(i)-fkb(i);
   if erreur_de_bit==0 NBE=NBE;
   else  NBE=NBE+1;
   end;
   BER(i)=NBE/i;

   %tex(:,i)=teta;
   %err_absolue_i(i) = norm(er_s)/(i);
   %err_relatif_i(i) = norm(er_s)/norm(s(1:i));

   semilogy(abs(e)), pause(0.001);

end;

tetaK=teta;
title=('variation de 1 erreur');
xlabel=('echantillons');
ylabel=('erreur=s(i)-fk(i)');
grid;

%************************************************************
%********* cycles de reconstitution sans adaptation *********
%************************************************************
m=0;
for i= ordre+1:1:l
flops(0);

   p_x=p(xb(i),xb(i-1),m1,m2);

   fk(i)= p_x'*tetaK;
```

```
   if fk(i)<0 fkb(i)=-1;
   else fkb(i)=1;
   end;
   erreur_de_bit=s(i)-fkb(i);
   if erreur_de_bit==0 NBE=NBE;
   else  NBE=NBE+1;
   end;
   BER(i)=NBE/i;

   er_s(i)=s(i) - fk(i);
   err_absolue_i(i) = norm(er_s)/(i);
   err_relatif_i(i) = norm(er_s)/norm(s(1:i));

end;

err_absolue = norm(er_s)/(1)
err_relatif = norm(er_s)/norm(s)


%************************************************************
%******************** liste des figures ********************
%************************************************************

figure(1);
plot(err_relatif_i);

figure(2);
plot(fk,'*');

figure(3);
plot(BER);

%************************************************************
%************ calcul du rapport signal/bruit **************
%************************************************************

s_b=20*log10(norm(s)/norm(bruit))
```

```
%************************************************************
% programme Matlab tenant compte de la quantification des
% différents variables. on utilise pour la quantification
% la fonction suivante:
% Quantif.m
%************************************************************
clear;
close all;
%************************************************************
%     INITIALISATION DES VARIABLES                          *
%************************************************************
lamda=0.91;
sigma=0.01;
ordre=150;  % nombre d'iteration necessaire pour l'adaptation du filtre
m=9;
q=4;
var_bruit=0.09;

for it=1:2:10
q=q+2;
clear bruit;
%*************************************************************
%*************** generation de signal carre ****************
%*************************************************************

s=rand(500,1);
l=length(s);
for i=1:l, if s(i)>=0.5, s(i)=1;, else s(i)=-1;, end, end

%*************************************************************
%***********generation de bruit de varience donnee *********
%*************************************************************

bruit=bruit(l,var_bruit);

%*************************************************************
%***************** effet du canal ***************************
%*************************************************************

for k=2:1:l

   x(k)=s(k)+0.5*s(k-1)-0.9*(s(k)+0.5*s(k-1))^3;
   xb(k)=x(k)+bruit(k);
end;

%*************************************************************
%***** initialisation des parametres de l'equalizeur *******
%*************************************************************

fk(1)=0;
teta=[];
   for k=1:1:(m*m)
   teta=[teta;0.1]; %(rand-0.5)/100];
   end;
```

```
tetaK_1=teta;
P=eye(m*m);
PK_1=P*sigma;
figure(4);

%************************************************************
%********* adaptation des parametres de l'equaliseur *******
%************************************************************

for i=2:1:ordre

   p_x=p(xb(i),xb(i-1),m,q);

   phi=p_x;

   PK=(1/lamda)*(PK_1-(PK_1*phi*((lamda+phi'*PK_1*phi)^(-
1))*(phi'*PK_1)));

   K=PK_1*phi*(lamda+phi'*PK_1*phi)^(-1);


   tetaK=tetaK_1+K*(s(i)-phi'*tetaK_1);

   fk(i)= p_x'*tetaK;

   er_s(i)=s(i) -fk(i);

   PK_1=PK;

   tetaK_1=tetaK;
   tex(:,i)=tetaK;


   semilogy(abs(er_s)), pause(0.001);

end;


%************************************************************
%********* cycles de reconstitution sans adaptation *********
%************************************************************
mx=0;
tetaKq=Qantif(tetaK,q,8);
sq=Qantif(s,q,2);
for k= ordre+1:1:l
xbq(k)=Qantif(xb(k),q,6);

   mx=mx+1;
   p_x=p(xbq(k),xbq(k-1),m,q);

   fk(k)= p_x'*tetaKq;
   fkq(k)=Qantif(fk(k),q,4);
   erreur_sq(mx)=sq(k) -fkq(k);
```

```
end;


end;
err_quadratique(it)=norm(erreur_sq)/norm(s)
err_absolueq(it) = norm(erreur_sq)/(l-ordre)


%****************************************
% calcul des valeurs du vecteur p
% en tenant conte de la quantification
%****************************************
function [p_x] = p(x1,x2,m,q);
den=0;
denq=0;
n=0;
num=0;

for m1=1:1:m

   for m2=1:1:m
      n=n+1;

      vx1=tri(x1,m1,m);
      vx2=tri(x2,m2,m);

      vx1q=Qantif(vx1,q,2);
      vx2q=Qantif(vx2,q,2);

      somme=vx1q*vx2q;
      sommeq=Qantif(somme,q,4);

      den=denq+sommeq;
      denq=Qantif(den,q,4);

      num(n)= vx1q * vx2q;
      numq(n)=Qantif(num(n),q,2);

   end;
end;
p_x1=numq/denq;
p_x1=Qantif(p_x1,q,2);

p_x=p_x1';
```

Annexes B

programme VHDL du réseau systolique utilisant PE_1
Annexe B1

Programme VHDL de PE_2
Annexe B2

```
-------------------------------------
-- Définition d'un package
-------------------------------------
-- Component : tdp1
--

library ieee;
use ieee.std_logic_1164.all;



PACKAGE tdp1 IS
      subtype vect_8 is std_logic_vector(7 downto 0);
      subtype vect_16 is std_logic_vector(15 downto 0);
      subtype vect_17 is std_logic_vector(16 downto 0);
      subtype vect_18 is std_logic_vector(17 downto 0);
      subtype vect_32 is std_logic_vector(31 downto 0);
      constant sd00 : vect_16 :=(others=>'0');
END tdp1 ;

-----------------------------------------------
-- programme VHDL pour le réseau de processeur
-----------------------------------------------

library fuzzy;
use fuzzy.tdp1.all;

ENTITY res_pe IS
   PORT (
            aa1,aa2,aa3 : IN vect_8;
            bb1,bb2,bb3 : IN vect_8;
            sn11,sn21,sn31 :IN vect_16;
      TETA11,TETA21,TETA22,TETA31,TETA32,TETA33,TETA41,TETA42,TETA51 :IN
vect_16;
      clk : IN bit;
      rst : IN bit;
            fk : OUT vect_16

   );
END res_pe;

---------------------------------------
ARCHITECTURE reseau OF res_pe IS
---------------------------------------
COMPONENT pe

            PORT (a,b:in vect_8;
                        sn_1,TETA:in vect_16;
                        clk,rst: in bit;
                        pa,pb:out vect_8;
                        sn :out vect_16);

END COMPONENT;
---------------------------------------
```

```
COMPONENT adder

            PORT (a,b:in vect_16;
                        clk,rst: in bit;
                        sn:out vect_16 );

END COMPONENT;

---------------------------------------

COMPONENT reg_8
      port ( vi :in vect_8;
                  clk : in bit;
                  rst : in bit;
            vo : out vect_8 );
end component;


---------------------------------------
 -- Internal Signals
---------------------------------------

   SIGNAL
a11_22,b11_21,a21_32,b21_31,a22_33,b22_32,a31_41,b32_41,a32_42,b33_42,a41
_51,b42_51:vect_8 ;

      SIGNAL sn11_22,sn21_32,sn22_33,sn31_41 : vect_16 ;
      SIGNAL sn32_42,sn33_2,sn41_51,sn42_4,sn51_6 : vect_16;

-- signaux internes non utiles

      SIGNAL b31,b41,b51,a33,a42,a51,a1,a2,a3,b1,b2,b3 : vect_8;

      SIGNAL spn24,spn46,spn6,sd0 : vect_16;


BEGIN
      sd0<=sd00;

   reg_a1 : reg_8
     PORT MAP (aa1,clk,rst,a1);

   reg_b1 : reg_8
      PORT MAP (bb1,clk,rst,b1);
   reg_a2 : reg_8
      PORT MAP (aa2,clk,rst,a2);

   reg_b2 : reg_8
      PORT MAP (bb2,clk,rst,b2);
   reg_a3 : reg_8
     PORT MAP (aa3,clk,rst,a3);

   reg_b3 : reg_8
      PORT MAP (bb3,clk,rst,b3);
```

```
   pe_11 : pe
      PORT MAP (a1,b1,sn11,TETA11,clk,rst,a11_22,b11_21,sn11_22);

   pe_21 : pe
      PORT MAP (a2,b11_21,sn21,TETA21,clk,rst,a21_32,b21_31,sn21_32);

   pe_22: pe
      PORT MAP (a11_22,b2,sn11_22,TETA22,clk,rst,a22_33,b22_32,sn22_33);

   pe_31 : pe
      PORT MAP (a3,b21_31,sn31,TETA31,clk,rst,a31_41,b31,sn31_41);

   pe_32 : pe
      PORT MAP
(a21_32,b22_32,sn21_32,TETA32,clk,rst,a32_42,b32_41,sn32_42);

   pe_33 : pe
      PORT MAP (a22_33,b3,sn22_33,TETA33,clk,rst,a33,b33_42,sn33_2);

   pe_41 : pe
      PORT MAP (a31_41,b32_41,sn31_41,TETA41,clk,rst,a41_51,b41,sn41_51);

   pe_42 : pe
      PORT MAP (a32_42,b33_42,sn32_42,TETA42,clk,rst,a42,b42_51,sn42_4);

   pe_51 : pe
      PORT MAP (a41_51,b42_51,sn41_51,TETA51,clk,rst,a51,b51,sn51_6);


      acc_2 : adder
            PORT MAP ( sn33_2,sd0,clk,rst,spn24);
      acc_4 : adder
            PORT MAP ( sn42_4,spn24,clk,rst,spn46);
      acc_6 : adder
            PORT MAP ( sn51_6,spn46,clk,rst,fk);

END reseau;


--*************************************************
--          Processeur elementaire
--*************************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;


entity pe is
      port (a,b :in vect_8;
            sn_1 : in vect_16;
            teta : in vect_16;
```

```
            clk : in bit;
            rst : in bit;
            pa :out vect_8;
            pb : out vect_8;
            sn : out vect_16 );
      end pe;

architecture behave of pe is

component adder_16_16
      port ( a,b :in vect_16;
            c : out vect_16 );
end component;

component mult_8_8
      port (      a,b : in vect_8;
            c : out vect_16 );
end component;

component mult_16_16
      port (      a,b : in vect_16;
            c : out vect_16 );
end component;

component reg_8
      port ( vi :in vect_8;
            clk : in bit;
            rst : in bit;
            vo : out vect_8 );
end component;

component reg_16
      port ( vi :in vect_16;
                  clk : in bit;
                  rst : in bit;
            vo : out vect_16 );
end component;

signal c,pr,prn,rsn :vect_16;
signal rpa,rpb : vect_8;

begin

mult1 : mult_8_8
      port map (a,b,c);

mult2 : mult_16_16
      port map (c,teta,pr);

adder : adder_16_16
      port map (sn_1,pr,prn);

reg1 : reg_8
      port map (a,clk,rst,pa);
```

```
reg2 : reg_8
      port map (b,clk,rst,pb);

--reg3 : reg_8
      --port map (rpa,clk,rst,pa);

--reg4 : reg_8
      --port map (rpb,clk,rst,pb);

reg5 : reg_16
      port map (prn,clk,rst,sn);

--reg6 : reg_16
      --port map (rsn,clk,rst,sn);
end behave;


--**************************************
--          multiplieur 8x8
--**************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;

Entity mult_8_8 IS
      port ( a,b : in vect_8;
            c : out vect_16 );
      end mult_8_8;

architecture behave of mult_8_8 is

      begin

            process(a,b)
variable reg_c : vect_16 := (others =>'0');
            begin
            reg_c := signed(a)*signed(b);
            c <= reg_c (15) & reg_c(13 downto 0) & ('0');
            end process;
       end behave;

--**************************************
--          multiplieur 16x16
--**************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;

Entity mult_16_16 IS
```

```
      port (      a,b : in vect_16 ;
            c : out vect_16 );
      end mult_16_16;

architecture behave of mult_16_16 is

      begin

            process(a,b)
            variable reg_c : vect_32 :=(others =>'0');
            begin
            reg_c := signed(a)*signed(b);
            c <= reg_c (31) & reg_c(29 downto 15);
            end process;
       end behave;

--************************************
--    additionneur 16 x 16
--************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;

entity adder_16_16 is
      port ( a,b :in vect_16;
            c : out vect_16 );
      end adder_16_16 ;

Architecture behave of adder_16_16 is

      begin
            process(a,b)
            begin
            c <= signed(a) + signed(b);
            end process;
      end behave;


--************************************
--    Registre 8
--************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;



entity reg_8 is
      port ( vi :in vect_8;
                  clk : in bit;
```

```
                rst : in bit;
            vo : out vect_8 );
      end reg_8;

architecture behave of reg_8 is

      begin
            process (vi,clk,rst)

            begin

            if (rst='1') then
                  vo<=(others => '0');
            elsif (clk='1') and (clk'event) then
                  vo<=vi;
            end if;
            end process;
      end behave;

--************************************
--    Registre 16
--************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;


entity reg_16 is
      port ( vi :in vect_16;
                  clk : in bit;
                  rst : in bit;
            vo : out vect_16 );
      end reg_16;

architecture behave of reg_16 is

      begin
            process (vi,clk,rst)

            begin
            if rst='1' then
                  vo<=(others => '0');
            elsif (clk='1') and (clk'event) then
                  vo<=vi;
            end if;
            end process;
      end behave;



--************************************
--    additionneur 16 x 16
```

```
--**************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;

library fuzzy;
use fuzzy.tdp1.all;

entity adder is
      port ( a,b :in vect_16;
                        clk:in bit;
                        rst:in bit;
                  sn : out vect_16 );
end adder;


--**********************************************
architecture behave of adder is
--**********************************************

component adder_16_16
      port ( a,b :in vect_16;
            c : out vect_16 );
end component;


component reg_16
      port ( vi :in vect_16;
                  clk : in bit;
                  rst : in bit;
            vo : out vect_16 );
end component;

signal ci : vect_16;

begin

adder1 : adder_16_16
      port map (a,b,ci);

reg1 : reg_16
      port map (ci,clk,rst,sn);

end behave;
```

```
--************************************
-- package servant à la modélisation
-- de PE_2
--************************************
--Mourad Zakhama 1998
--Universite du Quebec a Trois-Rivieres


LIBRARY ieee;
USE ieee.std_logic_1164.all;


--=======================================================
--PAQUETAGES DU PROGRAMME
--=======================================================

--*********************************
--PAQUETAGE DES TYPES ET CONSTANTES
--*********************************

PACKAGE constantes IS

--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  CONSTANT word:INTEGER :=8;     --DETERMINE LA LONGUEUR DANS
L'ARCHITECTURE
--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


TYPE sig_vect IS ARRAY(INTEGER RANGE <>) of std_logic_vector(word-1
downto 1);


END constantes;

--************************
--PAQUETAGE DES COMPOSANTS
--************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.tdp1.all;
use fuzzy.constantes.all;


PACKAGE arith IS

--******************
--    registre8
--******************
component reg_8
port ( vi :in vect_8;
                  clk : in std_logic;
```

```
                rst : in std_logic;
            vo : out vect_8 );
end component;

component reg_1
port ( vi :in std_logic;
                clk : in std_logic;
                rst : in std_logic;
            vo : out std_logic );
end component;
--******************
--COMPOSANTS DE BASE
--******************

COMPONENT cell                                          --Cellule CSA
PORT
       (a,b,c   :   in  std_logic;
        u,v     :   out std_logic);
END COMPONENT;
-------------------------------------------------------
COMPONENT ou_ex                        --XOR
 PORT(a,b:in std_logic;
      c:out std_logic);
end COMPONENT;
-------------------------------------------------------
COMPONENT et                           --AND
PORT(a,b:in std_logic;c:out std_logic);
END COMPONENT;
-------------------------------------------------------

component mux_8
      port(a,b: in std_logic_vector(word downto 1);
            c: out std_logic_vector(word downto 1);
            ctl : in std_logic );
end component;

component mux_1
      port(a,b: in std_logic;
            c: out std_logic;
            ctl : in std_logic );
end component;


COMPONENT inv_cell                                  --Cellule qui fait le
complement 1
  PORT(a,in_ctl :IN  std_logic;
       b,out_ctl:OUT std_logic);
END COMPONENT;

--*****************************
--ETAGES DE COMPOSANTS DE BASES
--*****************************

COMPONENT etage_mult                --Assemblage de cellule CSA
```

```
 GENERIC(nb_cell:INTEGER);

  PORT(
     a,b,c:in std_logic_vector(nb_cell downto 1);
     u,v:out std_logic_vector(nb_cell downto 1));
END COMPONENT;
-------------------------------------------------------
COMPONENT etage_et                    --Assemblage de AND
 GENERIC(nb_et:INTEGER);

 PORT(a   :IN  std_logic_vector(nb_et DOWNTO 1);
      b   :IN  std_logic;
      c   :OUT std_logic_vector(nb_et DOWNTO 1));
END COMPONENT;
-------------------------------------------------------

--*********************
--UNITES ARITHMETIQUES
--*********************

COMPONENT csa
PORT(
     a:IN std_logic_vector(word DOWNTO 1);
     b:IN std_logic_vector(word DOWNTO 1);
     u,v:OUT std_logic_vector(word-1 DOWNTO 1);
     sign_mult:out std_logic);
END COMPONENT;
-------------------------------------------------------
COMPONENT full_adder
PORT(
     a,b:in std_logic_vector(word downto 1);
     c:out std_logic_vector(word downto 1);
     Cin:IN std_logic);
END COMPONENT;
-------------------------------------------------------
COMPONENT comp_2
PORT(a:IN std_logic_vector(word-1 downto 1);
     ctl:IN std_logic;
     b:OUT std_logic_vector(word downto 1));
END COMPONENT;
-------------------------------------------------------
END arith;

--=====================================================
--Cellule de base pour le pipeline d'un MAC
--=====================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY cell IS

 PORT
       (a,b,c    :    in  std_logic;
```

```
        u,v     :   out std_logic);
END cell;

ARCHITECTURE behav OF cell IS
BEGIN

calcul:PROCESS(a,b,c)
BEGIN
 u<=a XOR b XOR c;
 v<=(a AND b)OR(a AND c)OR(b AND c);
END PROCESS;

END behav;



--=======================================================
--Porte ET (2 entrees)
--=======================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY et IS

PORT(a,b:in std_logic;c:out std_logic);

END et;

ARCHITECTURE behav OF et IS
BEGIN
c<=a and b;
END behav;

--=======================================================
--Porte XOR (2 entrees)
--=======================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ou_ex IS

PORT(a,b:in std_logic;c:out std_logic);

END ou_ex;

ARCHITECTURE behav OF ou_ex IS
BEGIN
c<=a xor b;
END behav;
--=======================================================
--VHDL pour l'assemblage d'une serie de cellules de base
--=======================================================
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.arith.all;


ENTITY etage_mult IS

  GENERIC(nb_cell:INTEGER);

  PORT(
     a,b,c:in std_logic_vector(nb_cell downto 1);
     u,v:out std_logic_vector(nb_cell downto 1));

END etage_mult;

ARCHITECTURE struct OF etage_mult IS

SIGNAL bidon:std_logic;

begin

G1:FOR i IN 1 to nb_cell GENERATE
 G2:IF i=1 GENERATE
  premier_cellule:cell
    port map(a(i),b(i),c(i),bidon,v(i));
 END GENERATE;

  G3:IF i>1 and i<=nb_cell GENERATE
  cellule:cell1
    port map(a(i),b(i),c(i),u(i-1),v(i));
 END GENERATE;
END GENERATE;

u(nb_cell)<='0';

END struct;


--=========================================================
--VHDL pour l'assemblage de ET logique
--=========================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.arith.all;
USE fuzzy.constantes.all;

ENTITY etage_et IS
GENERIC(nb_et:INTEGER);
PORT(a  :IN  std_logic_vector(nb_et DOWNTO 1);
     c  :OUT std_logic_vector(nb_et DOWNTO 1);
```

```
     b  :IN std_logic);
END etage_et;

ARCHITECTURE struct OF etage_et IS

BEGIN

G1:FOR i in 1 to nb_et GENERATE
et_logic:et
PORT MAP(a(i),b,c(i));

END GENERATE;

END struct;


--=========================================================
--MULTIPLIEUR PARAMETRISE
--=========================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.arith.all;
USE fuzzy.constantes.all;

ENTITY csa IS
PORT(
     a:IN std_logic_vector(word DOWNTO 1);
     b:IN std_logic_vector(word DOWNTO 1);
     u,v:OUT std_logic_vector(word-1 downto 1);
     sign_mult:out std_logic);
END csa;

ARCHITECTURE struct OF csa IS

--************************************
--DECLARATION DES SIGNAUX DE CONNECTION
--************************************

SIGNAL ai,bi,ci:sig_vect(word-1 DOWNTO 1);
SIGNAL zero:std_logic;
SIGNAL zeros:std_logic_vector(word-1 downto 1);

--************************
--DEBUT DE L'ARCHITECTURE
--************************

BEGIN
zero<='0';
zeros<=(others=>'0');
G1: FOR i IN 1 TO (word-2) GENERATE

  G2:IF i=1 GENERATE
```

```
  calcul_entree1:etage_et
     GENERIC MAP(word-1)
     PORT MAP(a=>b(word-1 downto 1),b=>a(i),c=>ai(i));
  calcul_entree2:etage_et
     GENERIC MAP(word-1)
     PORT MAP(a=>b(word-1 downto 1),b=>a(i+1),c=>bi(i));
  premier_etage:etage_mult
     GENERIC MAP(word-1)
     PORT MAP(ai(i),bi(i),zeros,ai(i+1),bi(i+1));
  signe_detector:ou_ex
     port map(a(word),b(word),sign_mult);
  END GENERATE;

  G3:IF i > 1 GENERATE
  calcul_entree3:etage_et
     GENERIC MAP(word-1)
     PORT MAP(b(word-1 downto 1),a(i+1),ci(i));
  etage_paire:etage_mult
     GENERIC MAP(word-1)
     PORT MAP(ai(i),bi(i),ci(i),ai(i+1),bi(i+1));
  END GENERATE;

END GENERATE;

u<=ai(word-1);
v<=bi(word-1);

END struct;



--=============================================================================
--Cellule d'inversion
--=============================================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.constantes.all;
USE fuzzy.arith.all;

ENTITY inv_cell IS
  PORT(a,in_ctl :IN  std_logic;
       b,out_ctl:OUT std_logic);
  END;

ARCHITECTURE struct OF inv_cell IS
BEGIN

 out_ctl<=in_ctl;
 ou_exclusif:ou_ex
  PORT MAP(a=>a,b=>in_ctl,c=>b);

END struct;
```

```
--=============================================================================
--VHDL pour effectuer le complement 2 si necessaire (mettre carry in du
FA a 1)
--=============================================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.constantes.all;
USE fuzzy.arith.all;

ENTITY comp_2 IS
PORT(a:IN std_logic_vector(WORD-1 downto 1);
     ctl:IN std_logic;
     b:OUT std_logic_vector(WORD downto 1));
END comp_2;

ARCHITECTURE behav OF comp_2 IS

SIGNAL propagation:std_logic_vector(WORD-1 downto 1);
BEGIN

G1:for i IN 1 TO word-1 GENERATE

 G2:IF i=1 GENERATE
   premiere_cellule:inv_cell
     PORT MAP(a=>a(i),b=>b(i),in_ctl=>ctl,out_ctl=>propagation(i));
 END GENERATE;

 G3:IF i>1 GENERATE
   cellule:inv_cell
     PORT MAP(a=>a(i),b=>b(i),in_ctl=>propagation(i-
1),out_ctl=>propagation(i));
 END GENERATE;

END GENERATE;

b(word)<=propagation(word-1);

END behav;

--=============================================================================
--VHDL pour le full adder
--=============================================================================

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
LIBRARY fuzzy;
USE fuzzy.constantes.all;

ENTITY full_adder IS
```

```
PORT(
     a,b:IN std_logic_vector(word downto 1);
     c:OUT std_logic_vector(word downto 1);
     Cin:IN std_logic);


END full_adder;

ARCHITECTURE struct OF full_adder IS

  SIGNAL temp:std_logic_vector(word DOWNTO 1);

BEGIN


  temp(1)<=Cin;
  temp(word downto 2)<="0000000";

  c<=unsigned(a)+unsigned(b)+unsigned(temp);

END struct;

--=========================================================================
--Multiplexeur 8std_logics, 2 entrees
--=========================================================================
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.constantes.all;

ENTITY mux_8 IS
  PORT( a,b:IN std_logic_vector(word DOWNTO 1);
        c  :OUT std_logic_vector(word DOWNTO 1);
        ctl:IN std_logic);
END mux_8;

ARCHITECTURE behav OF mux_8 IS
BEGIN

multiplex:PROCESS(a,b,ctl)
BEGIN

CASE(ctl) IS
  WHEN '0' => c<=a;
  WHEN '1' => c<=b;
  WHEN OTHERS=>NULL;
END CASE;

END PROCESS;

END behav;

--=========================================================================
--Multiplexeur 1std_logics, 2 entrees
```

```
--=====================================================================
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.constantes.all;

ENTITY mux_1 IS
  PORT( a,b:IN std_logic;
        c  :OUT std_logic;
        ctl:IN std_logic);
END mux_1;

ARCHITECTURE behav OF mux_1 IS
BEGIN

multiplex:PROCESS(a,b,ctl)
BEGIN

CASE(ctl) IS
  WHEN '0' => c<=a;
  WHEN '1' => c<=b;
  WHEN OTHERS=>NULL;

END CASE;

END PROCESS;

END behav;

--=====================================================================
--Registre 8std_logics avec reset
--=====================================================================

--**************************************
--    Registre 8
--**************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;



entity reg_8 is
      port ( vi :in vect_8;
                  clk : in std_logic;
                  rst : in std_logic;
            vo : out vect_8 );
      end reg_8;

architecture behave of reg_8 is

      begin
```

```
            process (vi,clk,rst)

            begin

            if (rst='1') then
                  vo<=(others => '0');
            elsif (clk='1') and (clk'event) then
                  vo<=vi;
            end if;
            end process;
      end behave;

--**************************************
--    Registre 1
--**************************************
library ieee;
use ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
library fuzzy;
use fuzzy.tdp1.all;



entity reg_1 is
      port ( vi :in std_logic;
                  clk : in std_logic;
                  rst : in std_logic;
            vo : out std_logic );
      end reg_1;

architecture behave of reg_1 is

      begin
            process (vi,clk,rst)

            begin

            if (rst='1') then
                  vo<= '0';
            elsif (clk='1') and (clk'event) then
                  vo<=vi;
            end if;
            end process;
      end behave;


--**********************************
-- processeur élémentaire PE2
--**********************************

--************************
--  controleur
--************************
library ieee;
```

```
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.arith.all;

entity controleur is
      port (clk,rst : in std_logic;
            ctl1,ctl2 : out std_logic);
end controleur;

architecture behave of controleur is

begin
      process(clk,rst)
variable cont : std_logic_vector (1 downto 0);
            begin
            if rst='1' then
                  cont:="00";
                  ctl1<='0';
                  ctl2<='0';
            elsif (clk='1') and (clk'event) then
                  case cont is
                        when "00" =>
                              cont:="01";
                              ctl1<='0';
                              ctl2<='0';
                        when "01" =>
                              cont:="10";
                              ctl1<='1';
                              ctl2<='0';
                        when others =>
                              cont:="00";
                              ctl1<='1';
                              ctl2<='1';
                  end case;

            end if;
      end process;
end behave;


--************************************
-- processeur elementaire
--************************************
library ieee;
USE ieee.std_logic_1164.all;
LIBRARY fuzzy;
USE fuzzy.tdp1.all;
USE fuzzy.arith.all;

entity pe is
      port(       a,b,teta: in vect_8;
            sn_1: in vect_8;
            clk,rst : in std_logic;
            pa,pb:out vect_8;
```

```
            sn : out vect_8);
end pe;


--************************************

architecture behavioral of pe is

--************************************

component controleur
      port  (clk,rst : in std_logic;
            ctl1,ctl2 : out std_logic
            );
end component;

signal      s1,s2,u_c2,v_c2,s_1,s_2,c,c_out,tetas:vect_8;
signal      ctl1,ctl2,clk_ctl2 : std_logic;
signal      sign_1r,sign_1,c_in: std_logic;
signal      u_csa,v_csa:std_logic_vector(6 downto 0);

begin

control : controleur
      port map (clk,rst,ctl1,ctl2);

clk_ctl2<=not ctl2;

reg_pa : reg_8
      port map (a,clk_ctl2,rst,pa);

reg_pb : reg_8
      port map (b,clk_ctl2,rst,pb);

reg_teta : reg_8
      port map (teta,clk_ctl2,rst,tetas);

muxx_1 : mux_8
      port map (tetas,a,s1,ctl1);

muxx_2 : mux_8
      port map (b,c_out,s2,ctl1);

csa_1 : csa
      port map (s1,s2,u_csa,v_csa,sign_1);

comp1 : comp_2
      port map (u_csa,sign_1,u_c2);

comp2 : comp_2
      port map (v_csa,sign_1,v_c2);

muxx_3 : mux_8
      port map (sn_1,u_c2,s_1,ctl2);
```

```
muxx_4 : mux_8
      port map (v_c2,c_out,s_2,ctl2);

reg_4 : reg_1
      port map (sign_1,ctl2,rst,sign_1r);

muxx_5 : mux_1
      port map (sign_1,sign_1R,c_in,ctl2);

fa_1  : full_adder
      port map (s_1,s_2,c,c_in);

reg_2 : reg_8
      port map (c,clk,rst,c_out);

reg_3 : reg_8
      port map (c,clk_ctl2,rst,sn);

end behavioral;
```

Annexes C

Programme Matlab pour la génération du forcefile

```
--**************************
-- programme Matlab permettant de générer le forcefile
-- du réseau systolique
--**************************

fid=fopen('/u/hping/zakhama/Mem_micro/gen_test/simulation.sim','w');
time=40;


fprintf(fid,'forc rst %2.0f\n',1);
fprintf(fid,'run %2.0f\n',time);
fprintf(fid,'forc rst %2.0f\n',0);
fprintf(fid,'run %2.0f\n',time);

for i=1:1:20

fprintf(fid,'\n');

fprintf(fid,'forc clk %2.0f\n',1);

fprintf(fid,'forc aa1 %2.0f\n',0);
fprintf(fid,'forc aa2 %2.0f\n',0);
fprintf(fid,'forc aa3 %2.0f\n',0);

fprintf(fid,'forc bb1 %2.0f\n',0);
fprintf(fid,'forc bb2 %2.0f\n',0);
fprintf(fid,'forc bb3 %2.0f\n',0);

fprintf(fid,'forc TETA11 %2.0f\n',0);
fprintf(fid,'forc TETA21 %2.0f\n',0);
fprintf(fid,'forc TETA31 %2.0f\n',0);
fprintf(fid,'forc TETA22 %2.0f\n',0);
fprintf(fid,'forc TETA32 %2.0f\n',0);
fprintf(fid,'forc TETA41 %2.0f\n',0);
fprintf(fid,'forc TETA33 %2.0f\n',0);
fprintf(fid,'forc TETA42 %2.0f\n',0);
fprintf(fid,'forc TETA51 %2.0f\n',0);


fprintf(fid,'forc sn11 %2.0f\n',0);
fprintf(fid,'forc sn21 %2.0f\n',0);
fprintf(fid,'forc sn31 %2.0f\n',0);


fprintf(fid,'run %2.0f\n',time);

fprintf(fid,'\n');

fprintf(fid,'forc clk %2.0f\n',0);
fprintf(fid,'run %2.0f\n',time);

end;

load variable3_30;
```

```
t=length(tetaK_n);

for i=1:1:t
   TETA(i)=bin2dec(conv2bin(tetaK_n(i),16));
end;

z= length(xb_n);
for i=2:1:z
   x=conv2bin(xb_n(i),8);
   y=conv2bin(xb_n(i-1),8);
   v1(i)=bin2dec(x);
   v2(i)=bin2dec(y);
end;

v1(1)=0;
v1(2)=0;

v2(1)=0;
v2(2)=0;




for i=3:1:z

fprintf(fid,'\n');

fprintf(fid,'forc aa1 %2.0f\n',v1(i));
fprintf(fid,'forc aa2 %2.0f\n',v1(i-1));
fprintf(fid,'forc aa3 %2.0f\n',v1(i-2));

fprintf(fid,'forc bb1 %2.0f\n',v2(i));
fprintf(fid,'forc bb2 %2.0f\n',v2(i-1));
fprintf(fid,'forc bb3 %2.0f\n',v2(i-2));

fprintf(fid,'forc sn11 %2.0f\n',0);
fprintf(fid,'forc sn21 %2.0f\n',0);
fprintf(fid,'forc sn31 %2.0f\n',0);


fprintf(fid,'forc TETA11 %2.0f\n',TETA(1));
fprintf(fid,'forc TETA22 %2.0f\n',TETA(2));
fprintf(fid,'forc TETA33 %2.0f\n',TETA(3));
fprintf(fid,'forc TETA21 %2.0f\n',TETA(4));
fprintf(fid,'forc TETA32 %2.0f\n',TETA(5));
fprintf(fid,'forc TETA42 %2.0f\n',TETA(6));
fprintf(fid,'forc TETA31 %2.0f\n',TETA(7));
fprintf(fid,'forc TETA41 %2.0f\n',TETA(8));
fprintf(fid,'forc TETA51 %2.0f\n',TETA(9));

fprintf(fid,'run %2.0f\n',10);

fprintf(fid,'forc clk %2.0f\n',1);
```

```
fprintf(fid,'run %2.0f\n',time);

fprintf(fid,'\n');
fprintf(fid,'force clk %2.0f\n',0);
fprintf(fid,'run %2.0f\n',time);

end;


--****************************************
-- fonction permettant de faire
-- la conversion d'une valeur analogique
-- sur un nombre de bits donné
--****************************************

function y=conv2bin(xa,b)

x=xa*2^(b-1);

if xa>=0

   res=dec2bin(x,b);
else

   v1=abs(x+1);
   r(1)='1';
   r(2:b)=(comp(dec2bin(v1,(b-1))));
   res=strcat(r);
end;
y=res;



%****************************************
% fonction permettant de faire la conversion
% decimale-décimale pour être compatible
% avec Qhsim
%****************************************

function y=dec2dec(x,b)

limite = 2^(b-1);
s=size(x);
line=s(1);
colon=s(2);

for k=1:1:line

   if x(k) > limite

      x0=dec2bin(x(k),b);
      x00=x0(2:b);
```

```
      x1=bin2dec(x00);
      x01=comp(dec2bin(x1-1,b-1));
      x2=bin2dec(x01);
      res(k)= -x2 *2^(-b+1);

   else

      res(k)=x(k)*2^(-b+1);

   end;
end;
y=res;


%*********************************
% fonction réalisant le complement à
% deux d'une valeur donné
%*********************************
function y=comp(x)

s = size(x);

for i=1:1:s(1)
   for k=1:1:s(2)
      if x(i,k)=='0'
         y(i,k)='1';
      elseif x(i,k)=='1'
         y(i,k)='0';
      end;
   end;
end;


%************************************************
% programme Matlab pour la génération du force file
% pour le réseau systolique simplifié 2x2
%************************************************


fid=fopen('/u/hping/zakhama/Mem_micro/gen_test/simulation.sim','w');
time=50;


fprintf(fid,'forc rst %2.0f\n',1);
fprintf(fid,'run %2.0f\n',time);
fprintf(fid,'forc rst %2.0f\n',0);
fprintf(fid,'run %2.0f\n',time);

load variable5_11;
t=length(tetaK_n);

for i=1:1:t
   TETA(i)=bin2dec(conv2bin(tetaK_n(i),16));
end;
```

```
z= length(xb_n);

vxk1(1)=0;
vxk2(1)=0;
vxk3(1)=0;
vxk4(1)=0;
vxk5(1)=0;

xk1(1)=0;
xk2(1)=0;
xk3(1)=0;
xk4(1)=0;
xk5(1)=0;

vxk1(2)=0;
vxk2(2)=0;
vxk3(2)=0;
vxk4(2)=0;
vxk5(2)=0;

xk1(2)=0;
xk2(2)=0;
xk3(2)=0;
xk4(2)=0;
xk5(2)=0;

for i=5:1:z

vxk1(i)=bin2dec(conv2bin(tri(xb_n(i),1,5,8),8));
vxk2(i)=bin2dec(conv2bin(tri(xb_n(i),2,5,8),8));
vxk3(i)=bin2dec(conv2bin(tri(xb_n(i),3,5,8),8));
vxk4(i)=bin2dec(conv2bin(tri(xb_n(i),4,5,8),8));
vxk5(i)=bin2dec(conv2bin(tri(xb_n(i),5,5,8),8));


xk1(i)=bin2dec(conv2bin(tri(xb_n(i-1),1,5,8),8));
xk2(i)=bin2dec(conv2bin(tri(xb_n(i-1),2,5,8),8));
xk3(i)=bin2dec(conv2bin(tri(xb_n(i-1),3,5,8),8));
xk4(i)=bin2dec(conv2bin(tri(xb_n(i-1),4,5,8),8));
xk5(i)=bin2dec(conv2bin(tri(xb_n(i-1),5,5,8),8));

%a1=vxk1(i);
%a2=vxk2(i);
%a3=vxk3(i);

%b1=xk1(i);
%b2=xk2(i);
%b3=xk3(i);


fprintf(fid,'\n');

fprintf(fid,'forc clk %2.0f\n',1);
```

```
fprintf(fid,'forc aa1 %2.0f\n',xk1(i));
fprintf(fid,'forc aa2 %2.0f\n',xk2(i-1));
fprintf(fid,'forc aa3 %2.0f\n',xk3(i-2));
fprintf(fid,'forc aa4 %2.0f\n',xk2(i-3));
fprintf(fid,'forc aa5 %2.0f\n',xk3(i-4));


fprintf(fid,'forc bb1 %2.0f\n',vxk1(i));
fprintf(fid,'forc bb2 %2.0f\n',vxk2(i-1));
fprintf(fid,'forc bb3 %2.0f\n',vxk3(i-2));
fprintf(fid,'forc bb4 %2.0f\n',vxk2(i-3));
fprintf(fid,'forc bb5 %2.0f\n',vxk3(i-4));


fprintf(fid,'forc sn11 %2.0f\n',0);
fprintf(fid,'forc sn21 %2.0f\n',0);


fprintf(fid,'forc TETA1 %2.0f\n',TETA(1));
fprintf(fid,'forc TETA2 %2.0f\n',TETA(2));
fprintf(fid,'forc TETA3 %2.0f\n',TETA(3));
fprintf(fid,'forc TETA4 %2.0f\n',TETA(4));
fprintf(fid,'forc TETA5 %2.0f\n',TETA(5));
fprintf(fid,'forc TETA6 %2.0f\n',TETA(6));
fprintf(fid,'forc TETA7 %2.0f\n',TETA(7));
fprintf(fid,'forc TETA8 %2.0f\n',TETA(8));
fprintf(fid,'forc TETA9 %2.0f\n',TETA(9));
fprintf(fid,'forc TETA10 %2.0f\n',TETA(10));
fprintf(fid,'forc TETA11 %2.0f\n',TETA(11));
fprintf(fid,'forc TETA12 %2.0f\n',TETA(12));
fprintf(fid,'forc TETA13 %2.0f\n',TETA(13));
fprintf(fid,'forc TETA14 %2.0f\n',TETA(14));
fprintf(fid,'forc TETA15 %2.0f\n',TETA(15));
fprintf(fid,'forc TETA16 %2.0f\n',TETA(16));
fprintf(fid,'forc TETA17 %2.0f\n',TETA(17));
fprintf(fid,'forc TETA18 %2.0f\n',TETA(18));
fprintf(fid,'forc TETA19 %2.0f\n',TETA(19));
fprintf(fid,'forc TETA20 %2.0f\n',TETA(20));
fprintf(fid,'forc TETA21 %2.0f\n',TETA(21));
fprintf(fid,'forc TETA22 %2.0f\n',TETA(22));
fprintf(fid,'forc TETA23 %2.0f\n',TETA(23));
fprintf(fid,'forc TETA24 %2.0f\n',TETA(24));
fprintf(fid,'forc TETA25 %2.0f\n',TETA(25));


fprintf(fid,'run %2.0f\n',time);

fprintf(fid,'\n');
fprintf(fid,'forc clk %2.0f\n',0);
fprintf(fid,'run %2.0f\n',time);

end;
```

Annexe D

Programme C pour l'implantation physique sur le FPGA

```
//************************************
// programme c permettant d'implanter
// un design *.bit dans le FPGA
// Mourad Zakhama
// 10-10-99
//************************************

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include "mtrci.h"

int main(int argc, char *argv[])
{
   HANDLE hDevice;
   BOOL status;

   // print banner

   printf( "\nTest application - reseau\n\n" );

   // open reconfigurable computing device
   hDevice = RciOpenDevice(NULL , "#0",0);
   if( hDevice == NULL )
   {
      printf( "Test: open failed, error = 0x%08x\n", RciGetLastError());
      return 1;
   }

   // configure VPE with reseau core

   printf( "Press any key to configure VPE with reseau core.\n" );
   getch();

   status = RciBootDevice(hDevice, RCI_DEV_VPE | RCI_DEV_DMA0,
"reseau.bit");
   if( !status )
   {
      printf( "Test: VPE boot failed, error = 0x%08x\n",
RciGetLastError());
      return 1;
   }
   printf( "you design is implemented on VPE \n\n" );

   // set the VPE clock at 20MHz
   printf( "Press any key to configure the VPE clk at 20MHz.\n" );
   getch();
   status = RciSetDeviceClock(hDevice, RCI_CLK_ON | RCI_CLK_VPE, 20000);
   if( !status )
   {
```

```
      printf( "Test: VPE clk failed, error = 0x%08x\n",
RciGetLastError());
      return 1;
   }
   printf( "The VPE clk is set at 20MHz \n\n" );

   // clear VPE configuration.

   printf( "Press any key to clear the VPE configuration.\n" );
   getch();
   status = RciBootDevice(hDevice, RCI_DEV_VPE | RCI_DEV_DMA0, NULL);
   if( !status )
   {
      printf( "Test: VPE boot failed, error = 0x%08x\n",
RciGetLastError());
      return 1;
   }
   printf( "the configuration of your VPE is cleared.\n\n" );

   // close reconfigurable computing device
   status = RciCloseDevice(hDevice);
   if( !status )
   {
      printf( "Test: close failed, error = 0x%08x\n", RciGetLastError());
      return 1;
   }

   return 0;
}
```

Annexes E

Publication réalisé à la conférence IEEE (CCECE'99)
Canadian Conference on Electrical and Computer Engineering
Edmonton, Alberta, Canada, 9-12 may, 1999

IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'99)
Edmonton, Alberta, Canada, 9-12 may, 1999

A Systolic Architecture for Channel Equalization Based on a Piecewise Linear Fuzzy Logic Algorithm

Mourad Zakhama and Daniel Massicotte

Electrical Engineering Department, Université du Québec à Trois-Rivières
Research Group on Industrial Electronics
C.P. 500, Trois-Rivières, Québec, Canada, G9A 5H7
Tel.: 1-(819)-376-5071, Fax : 1-(819)-376-5219
E-mail : {Mourad_Zakhama, Daniel_Massicotte}@uqtr.uquebec.ca

Abstract

A systolic architecture dedicated to a piecewise linear fuzzy logic algorithm for a nonlinear channel equalization is presented. The piecewise linear membership function proposed for the inference step is more suitable for a VLSI implementation than the Gaussian function proposed in the literature. Depending on the number of piecewise linear membership functions (m) on the input space, the equalizer can perform for linear or nonlinear channel equalization. The performance evaluation of the systolic architecture is evaluated in terms of speed and area. The latency and throughput of the 16-bits design are respectively $(2m+1)f_c$ and f_c, where f_c is the clock frequency. In 0.5 µm CMOS technology the f_c is evaluated at 40 MHz.

1. Introduction

In numerical communications, messages are composed of a succession of symbols. The objective is to convey these messages, *s*, from the transmitter to the receiver [9]. This is done via a data communication channel, which represents the physical environments by which signals pass (Fig. 1). Each physical environment has its own linear and nonlinear characteristics, which induce deformations on signals, giving an output signal completely different from the transmit messages. As a result, in each medium of transmission, an additive noise η is inevitable and the exit signal, $\tilde{y}$, is deformed and disturbed. In a nonlinear channel equalization is to solve the reconstruction problem which consists of a regularized inversion of an operator

$$\{\hat{x}(n)\} = \Re[\{\tilde{y}(n)\}, \Theta] \quad (1)$$

where $\Re$ is an operator of reconstruction and Θ represents the vector of the parameters. The measurement of $\tilde{y}(n)$ leads us to deduce the value of the estimated entry $x(n)$ through a corrective processor called a channel equalizer that makes it possible to obtain $\hat{x}(n)$.

Many algorithm propositions of channel equalization can be found in the literature. Some are for a linear [6], [9], and others are for a nonlinear [1], [3]. A fuzzy equalizer based on a fuzzy adaptive filter was proposed in [1], and an equalizer based on a fuzzy system was proposed in [3]. However, there are few VLSI architectures dedicated to nonlinear channel equalization. In [4] we find an architecture for bayesian equalization using fuzzy filters.

The fuzzy logic-based algorithm for nonlinear channel equalization is described in Section 2. In Section 3, we proposed the systolic architecture. The simulation results of the piecewise linear fuzzy logic algorithm and the performance evaluation of the proposed architecture in 0.5 µm CMOS technology for linear and nonlinear channels was presented in Section 5. A conclusion is given in Section 6.

2. Fuzzy Logic Algorithm for Channel Equalization

Three steps are used to define the fuzzy algorithm proposed in [1]: the fuzzification step, the optimization of inferences rules and the defuzzification step. The first step is to assign to each input sample, $\tilde{y}_k$, a degree of membership defined by a membership function, μ, which covers the input space. This determines the fuzzification block. The second step consists of using an adaptive algorithm (e.g., LMS, RLS algorithms) which makes it possible to build the rules of inference by optimizing the error between the result of correction $\hat{x}$ and the ideal signal *s*. This

This work was supported by the Natural Science and Engineering Research Council of Canada and by Le Fonds FCAR Québec.

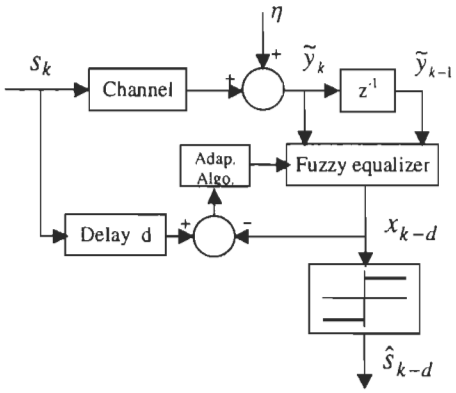


Figure 1 : Schematic bloc of channel equalization

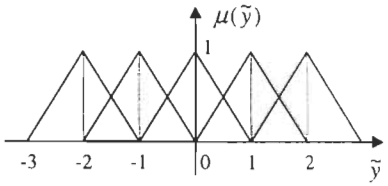


Figure 2: Piecewise Linear membership function with m=5 $\bar{y}^{l_j}$ =-2, -1, 0, 1 ,2 for l_j=1, 2, ..., m_j and j=1,2.

procedure gives the vector of parameters Θ. The third step consists of making a decision based on these rules, which constitutes the defuzzification block. The output of these blocks provides an estimated value $\hat{x}_k$ as follows

$$\hat{x}_k = \boldsymbol{p}^T(\tilde{\boldsymbol{y}})\Theta \tag{2}$$

The vector $\boldsymbol{p}(\tilde{\boldsymbol{y}})$ is a vector generated on the level of inference blocks given by the following relations

$$\boldsymbol{p}(\tilde{\boldsymbol{y}}) = \begin{bmatrix} p^{(1,1)}(\tilde{\boldsymbol{y}}) & p^{(2,1)}(\tilde{\boldsymbol{y}}) & \cdots & p^{(m_1,1)}(\tilde{\boldsymbol{y}}); \cdots \\ p^{(1,2)}(\tilde{\boldsymbol{y}}) & p^{(2,2)}(\tilde{\boldsymbol{y}}) & \cdots & p^{(m_1,2)}(\tilde{\boldsymbol{y}}); \cdots \\ p^{(1,m_2)}(\tilde{\boldsymbol{y}}) & p^{(2,m_2)}(\tilde{\boldsymbol{y}}) & \cdots & p^{(m_1,m_2)}(\tilde{\boldsymbol{y}}) \end{bmatrix}^T \tag{3}$$

with l_j=1,2,...,m_j, j=1,2, $\tilde{\boldsymbol{y}} = [\tilde{y}_k \;\; \tilde{y}_{k-1}]^T$ and dim($\boldsymbol{p}$)=(m_1×m_2)×1. In this equation, m_1 is the number of membership functions covering the input space of $\tilde{y}_k$ and m_2 is the number of membership functions covering the input space of $\tilde{y}_{k-1}$. The elements of the vector $\boldsymbol{p}(\tilde{\boldsymbol{y}})$ are computed as follows:

$$p^{(l_1,l_2)}(\tilde{\boldsymbol{y}}) = \frac{\mu(\tilde{y}_k, l_1)\mu(\tilde{y}_{k-1}, l_2)}{c_k(\tilde{\boldsymbol{y}})} \tag{4}$$

and the none null denominator

$$c_k(\tilde{\boldsymbol{y}}) = \sum_{l_1=1}^{m_1}\sum_{l_2=1}^{m_2}\mu(\tilde{y}_k, l_1)\mu(\tilde{y}_{k-1}, l_2) \tag{5}$$

where $\mu(\tilde{\boldsymbol{y}}, l_j)$ defines the result of the fuzzification step. It expresses the membership of the vector $\tilde{\boldsymbol{y}}$ to the membership function of l_j. A *Gaussian* membership function is proposed in [1]

$$\mu(\tilde{y}_{k-i+1}) = \exp\left[-\frac{1}{2}\left(\tilde{y}_{k-i+1} - \bar{y}^{l_j}_{k-i+1}\right)^2 \Big/ \sigma_i^{2l}\right] \tag{6}$$

where i=1,2 and $\bar{y}^{l_j}_{k-i+1}$ and σ_i^l are empirical parameters. The digital implementation of this function is difficult for a large number of membership functions. As regards digital implementation, the piecewise linear function is preferable for obtaining a parallel architecture with a high throughput. We proposed using a piecewise linear fuzzy logic algorithm defined as follows

$$\mu(\tilde{y}_{k-i+1}) = \begin{cases} 1 - \dfrac{\left|\tilde{y}_{k-i+1} - \bar{y}^{l_j}_{k-i+1}\right|}{\sigma_i^l} & for \left|\tilde{y}_{k-i+1} - \bar{y}^{l_j}_{k-i+1}\right| < \sigma_i^l \\ 0 & elsewhere \end{cases} \tag{7}$$

to reconstruct the input signal of a linear or a nonlinear channel. An example of the piecewise linear membership function is shown in Fig. 1.

The vector Θ of Eq. (2) represents the vector of parameters adapted to the channel by an algorithm such as RLS algorithm [1], [2]. This vector is defined as follows

$$\Theta = \begin{bmatrix} \theta^{(1,1)} & \theta^{(2,1)} & \cdots & \theta^{(m_1,1)}; \cdots \\ \theta^{(1,2)} & \theta^{(2,2)} & \cdots & \theta^{(m_1,2)}; \cdots \\ \theta^{(1,m_2)} & \theta^{(2,m_2)} & \cdots & \theta^{(m_1,m_2)} \end{bmatrix}^T \tag{8}$$

with dim(Θ)=(m_1×m_2)×1. From the estimated result obtained from Eq. (2), a decision function is applied to obtain the input estimate of the message as follows

$$\hat{s}_k = sign(\hat{x}_k) \tag{9}$$

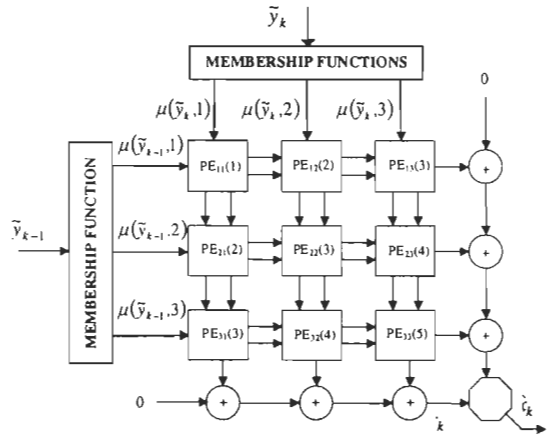


Figure 3: Systolic architecture for channel equalization based on the fuzzy logic with m_1=m_2=3.

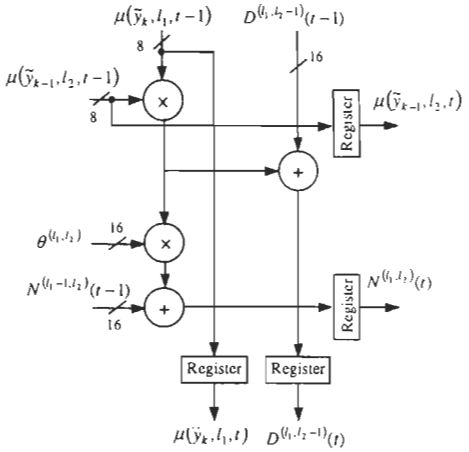


Figure 4 - Architecture of an elementary processor

For a stationary and invariant channel, these parameter vectors can be calculated in advance with an adaptive algorithm. $\hat{x}(n)$ is a scalar corresponding to the equalizer output and is the result carried out from defuzzification by the centroid method [1]. In the next section, we propose a VLSI systolic architecture dedicated to a system based on the equations described above for the channel equalization.

3. Proposed Systolic Architecture

The study of the piecewise linear fuzzy logic algorithm structure and its simplification enable us to produce a highly parallel architecture. In the first step of digital implementation, we consider the case of a non-linear stationary channel where we have access to the vector Θ. Based on equations (2), (4), and (8) we can write

$$\hat{x}_k = \frac{\sum_{l_1=1}^{m_1}\sum_{l_2=1}^{m_2}\theta^{(l_1,l_2)}\mu(\tilde{y}_k,l_1)\mu(\tilde{y}_{k-1},l_2)}{c_k(\tilde{y})} \qquad (10)$$

We observed from this equation that the algorithm revealed the presence of independent recurrent equations, which constitute the foundation of systolic architectures [5], [7]. Fig. 3 shows the proposed systolic architecture for m_1=m_2=3. The membership functions defined by Eq. (7) cover the input space of $\tilde{y}_k$ and $\tilde{y}_{k-1}$. In Fig. 3, the data $\mu(\tilde{y}_k,l_1)$ and $\mu(\tilde{y}_{k-1},l_2)$ represents the membership of $\tilde{y}_k$ and $\tilde{y}_{k-1}$ respectively for each membership function of the input space.

The systolic architecture is composed of $m_1 \times m_2$ square processing elements (PE), m_1+m_2 adder units and one octagonal cell used for the division operation of Eq. (10) and the saturation function of Eq. (11). At each clock cycle t the $PE_{l_1,l_2}(t)$, shown in Fig. 4, must calculate the partial sum of the numerator and the denominator, $c_k(\tilde{y})$, of Eq. (10) as follows

$$N^{(l_1,l_2)}(t) = \theta^{(l_1,l_2)}\mu(\tilde{y}_k,l_1,\mathrm{t}-1)\mu(\tilde{y}_{k-1},l_2,\mathrm{t}-1) + N^{(l_1-1,l_2)}(t-1) \qquad (11)$$

$$D^{(l_1,l_2)}(t) = \theta^{(l_1,l_2)}\mu(\tilde{y}_k,l_1,\mathrm{t}-1)\mu(\tilde{y}_{k-1},l_2,\mathrm{t}-1) + D^{(l_1,l_2-1)}(t-1) \qquad (12)$$

where N and D are the partial sum of the numerator and the denominator respectively, coming from the PE_{l_1,l_2-1} and PE_{l_1-1,l_2} respectively. The contents of the registers are connected to the neighborhood processors in the next clock cycle t+1.

On the south and east boundaries of the processor array there are adders which calculate the partial sum of the numerator and the denominator respectively. Once the processor array is full, at each clock cycle t, the numerator and the denominator of Eq. (10) are available in the input of the octagonal cell.

The last step is to divide the numerator by the value $c_k(\tilde{y})$, which gives the estimation $\hat{x}_k$. However, the performance of this architecture is limited in frequency by the divider rate. If we look at Eq (5), we notice that the denominator is always positive and none null. The goal of the division is to obtain a result closer to {-1,1} pairs. Consequently we can use Eq. (9) without the deviser, and we can omit computing the value of

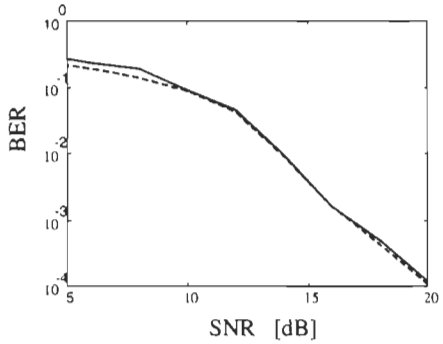


Figure 5: Robustness of noise for the nonlinear channel for both membership functions, the *Gaussian* function in dash line and the piecewise linear function in continuous line.

$c_k(\tilde{y})$ and the divider unit. Finally we obtain the output $\hat{s}_k$ from Eq. (9).

4. Simulation Results and Performance Evaluation

The method of correction proposed for the channel equalization was studied using synthetic data generated according to the following formulas:

$$\tilde{y}_k = s_k + 0.5s_{k-1} + \eta_k \quad (13)$$

for the linear channel and

$$\tilde{y}_k = s_k + 0.5s_{k-1} - 0.9(s_k + 0.5s_{k-1})^3 + \eta_k \quad (14)$$

for the nonlinear channel.

The input signal s_k is assumed to be an independent sequence taking values from {-1,1} with equal probability, and η_k denotes the zero mean white Gaussian noise for obtaining a signal noise ratio (SNR) of 5 dB to 20 dB for the study. The quality of correction was assessed using the bit error rate (BER). Fig. 5 illustrates robustness of noise for the nonlinear channel according to the both membership functions defined by the Eqs. (6) and (7) with m_j=5, $\bar{y}^{l_j}$ =-2,-1,0,1,2 for l_j=1,2,...,m_j and j=1,2 and σ_i^l=0.3 in the both cases. Each algorithm used 100 iterations for the adaptive step to obtain the vector Θ using the RLS algorithm proposed in [1], and 1000 data were used to calculate the BER. Fig. 5 illustrates the average of 10 repetitions.

The study of the quantification enabled us to conclude that 16-bits are sufficient to represent the internal variables of architecture presented in Fig.3 and the input variables, vector $\tilde{y}_k$, are represented on 8-bits.

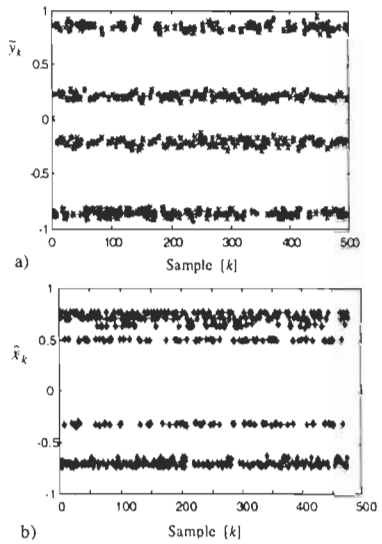


Figure 6 Test of correction for a linear channel with SNR=20 dB: a) the output signal $\tilde{y}_k$ and b) the correction $\hat{x}_k$ result with VHDL simulation (BER=0).

The performance evaluation is carried out in terms of the latency and throughput for a 16-bit wordlength for variables and an 8-bit wordlength for input samples. The design was made by means of standard CAD tools available from the Canadian Microelectronics Corporation (CMC). The structural model of the proposed architecture was made in VHDL using Mentor-Graphics CAD tools for register transfer level (RTL) modeling and simulation. Fig. 6 shows that the simulation of the VHDL code of the systolic architecture described in Fig.3 was carried out with the linear channel according to Eq. (13). For a nonlinear channel of Eq. (14) we need m_j>3.

The latency of the proposed architecture is evaluated at (m1+m2)+1 clock cycles. An important advantage, of this equalizer is that, once the network is full it can provide a result on each cycle. A low-effort synthesis optimization was made with Synopsys tools with the Hewlett-Packard 0.5-µm CMOS technology available from MOSIS through CMC. The integration area is about 7 000 transistors for the PE and the total number of transistors for m_1=m_2=3 is evaluated at 80 000 transistors including all blocs of the processor shown in Fig. 3. The clock frequency, f_c, of the architecture is evaluated to 40 MHz and is limited by the presence of two consecutive multipliers in each PE. We can increase this speed by the insertion of a register between both multipliers to create a pipeline stage inside the PE.

5. Conclusion

This paper described a piecewise linear fuzzy logic architecture dedicated to channel equalization. We have shown that the results of correction from using a piecewise linear membership function of a *Gaussian* function are relatively equivalent. The systolic architecture is composed of a processing element implemented in a square array to exploit the parallelism. The algorithm reveals the presence of null terms in the inference step according to Eq. (4), which can be eliminated to reduce the number of processing elements and consequently the surface of integration. Finally, the future work is to introduce an adaptive algorithm to implement in VLSI structure an nonlinear adaptive equalizer.

References

[1] L.-X. Wang and J. M. Mendel, "Fuzzy Adaptative Filter With Application to Nonlinear Channel Equalization", *IEEE Transactions on Fuzzy System*, Vol. 1, No. 3, August 1993, pp. 161-170.

[2] L.-X. Wang and J. M. Mendel, "Fuzzy Basis Functions, Universal Approximation, and Orthogonal Least Squares Learning", *IEEE Transactions on Neural Networks*, Vol. 3, No. 5, 1992, pp. 807-814.

[3] P. Sarwal and M. D. Srinath "Afuzzy logic system for channel equalization" *IEEE Transactions on Fuzzy Systems*, vol. 3, No. 2, May 1995.

[4] Sarat Kumar Patra and Bernard Mulgrew, "Efficent architecture for bayesian equalization using fuzzy filters", *IEEE Transactions on circuit and systems-II: Analogue and digital signal processing*, vol. 45, No. 7, July 1998.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to parallel computing, Design and analysis of algorithms".

[6] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, 1996.

[7] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays", *Journal of VLSI Signal Processing*, Vol. 1, No 2, October 1989, pp. 95-113.

[8] X. Liu, T. Adah and L. Demirekler, "A Piecewise Linear Recurrent Neural Network Structure and its Dynamics", *IEEE International Conference on Acoustics, Speech and Signal Processing*, Seattle, 1998, pp.1221-1224.

[9] J.G. Proakis, *Digital Communications*, 3rd Ed., McGraw-Hill, 1995.